CS-TR-94-13

Interval Linear Constraint Solving in Constraint Logic Programming

Chong-kan Chiu

Department of Computer Science

The Chinese University of Hong Kong

Hong Kong

December 1994

Interval Linear Constraint Solving in Constraint Logic Programming

by
Chong-kan Chiu

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Master in Philosophy
in the Division of Computer Science

The Chinese University of Hong Hong
December 1994

(This document is a reformatted version of the dissertation, and equivalent in contents.)

Abstract

Existing interval constraint logic programming languages, such as BNR Prolog, work under the framework of interval narrowing and are deficient in solving general systems of constraints over real, which constitute an important class of problems in engineering and other applications. In this thesis, we suggest to separate linear constraint solving from non-linear constraint solving. Two implementations of an efficient interval linear equality constraint solver, which are based on generalized interval Gaussian elimination and the incremental preconditioned interval Gauss-Seidel method, are proposed. We show how the solvers can be adapted to incremental execution and incorporated into a constraint logic programming language already equipped with a non-linear solver based on interval narrowing. The two solvers share common interval variables, interact and cooperate in a round-robin fashion during computation, resulting in an efficient interval constraint arithmetic language CIAL. The CIAL prototypes, based on CLP($\mathcal{R}$), are constructed and compared favourably against several major interval constraint logic programming languages.

Declaration

The work in this thesis is independent and original work of the author, except where explicit reference to the contrary has been made. No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or institution of education.

Author: ______________________________
Chong-kan Chiu
December 25, 1994

Acknowledgements

There are always a number of people that have helped out in one way or another. I am very grateful to all of them. There are some whom I would like to give special thanks.

My warmest thanks to my supervisor Dr. Jimmy Ho-man Lee, who has given invaluable support in developing these ideas. His faith in my approach kept my spirit alive during many months of darkness. He read several drafts of this thesis indefatigably, page to page, and provided me comments which went to change my presentation.

I hardly know how to acknowledge Tak-wai Lee. He has taken time away from his research to help and talk with me at all hours. I thanks for numerous inspiring discussions we had.

I have benefitted from interactions with the roommates in my office, including Siu-man Hsieh, Kwong-ip Liu, and Anthony Yiu-cheung Tang. They have provided an intellectual atmosphere and have made my life happy here.

Thanks to my colleagues for their good company. Here I include Chung-yuen Li, Chi-lok Chan, and Bobby Bo-ming Tong. But I do always remember the smiles of the rest.

My gratefulness further goes to Dr. Ho-fung Leung, Dr. Alex Siu-chi Hsu, and Prof. Spiro Michaylov, who provided many constructive comments on this thesis.

I am indebted to Prof. Joxan Jaffar, Prof. R. Baker Kearfott, and Dr. Roland Yap for e-mail discussions. Access to CLP($\mathcal{R}$), BNR Prolog, CLP(BNR), ICL, and Echidna is also gratefully acknowledged.

This work would not have been possible without generous financial support. I thank The Chinese University of Hong Kong for offering me Postgraduate Studentship in the past two years.

This thesis is dedicated to my family, who encourage me throughout these years. Without you, I would not have had the stamina to go though all these hard days.

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

Current status of Prolog arithmetic suffers from two deficiencies. First, the system predicate "is" [56] is functional in nature. It is incompatible with the relational paradigm of logic programming. Second, real numbers are approximated by floating-point numbers. Roundoff errors induced by floating-point arithmetic destroy the soundness [41] of computation. The advent of constraint logic programming [31] presents a solution to the first problem but the implementation of CLP languages, such as CLP($\mathcal{R}$) [32], are mostly based on floating-point arithmetic. The second problem remains.

The languages CAL [2] and RISC-CLP($\mathcal{R}$) [30] use symbolic algebraic methods to refrain from floating-point operations. Algebraic methods guarantee the soundness of numerical computation but they are time-consuming.

Previous efforts in the sub-symbolic camp, such as BNR Prolog [51], employ interval methods [45] and belong to the family of consistency techniques [42]. The main idea is to narrow the set of possible values of the variables of arbitrary real constraints using approximations of arc-consistency [10]. We collectively call these techniques *interval narrowing*. Interval narrowing has been shown to be applicable to critical path scheduling [51], X-ray diffraction crystallography [53], boolean constraint solving [9], and disjunctive constraint solving [9, 54]. However, interval narrowing is deficient in handling systems of linear constraints over real domain.

For example, interval narrowing fails to solve such simple systems as "$\{X + Y = 5, X - Y = 6\}$." Cleary [16] proposes a form of case analysis technique [57], *domain splitting*, as a remedy. Domain splitting partitions an interval into two, visits one, and visits the other upon backtracking. This backtracking tree search is expensive to perform. Furthermore, interval narrowing may sometimes fail or take a long time to detect inconsistency of linear systems. Thus, interval narrowing is opted for improvement in terms of efficiency.

Our work is motivated by the inadequacy of interval narrowing for interval linear constraint solving. The goal is to design a *sound* and *efficient* interval linear constraint solving method for CLP languages. We suggest to separate linear equality constraint solving from inequality and non-linear constraint solving. This separation calls for an employment of two constraint solvers: a linear solver and a non-linear solver. The linear solver consists of symbolic transformation and numerical method. The symbolic transformation of constraints helps to achieve a global analysis of the system of constraints; while variables are narrowed by the numerical method. The precisions of solutions and the speed of convergence are improved with the cooperation of the two techniques. The non-linear solver employs interval narrowing with splitting to solve inequalities and non-linear constraints.

1.1 Related Work

Prolog III [17], CAL [2], and RISC-CLP($\mathcal{R}$) [30] use symbolic algebraic methods to solve arithmetic constraints. Prolog III[1] employs a simplex algorithm to handle arithmetic over rational numbers. CAL computes over two domains: the real numbers and Boolean algebra with symbolic values. Constraints are solved by using Buchberger algorithm for computing Gröbner bases [11]. RISC-CLP($\mathcal{R}$) deals with non-linear arithmetic constraints by using Gröbner basis and Partial Cylindrical Algebraic Decomposition [29, 12].

In the sub-symbolic camp, Cleary [16] introduces "logical arithmetic," a relational version of interval arithmetic, into Prolog. He describes distinct algorithms, one for each kind of constraint over intervals, that narrow intervals associated with a constraint by removing values that do not satisfy the constraint. A constraint relaxation cycle is needed to coordinate the execution of the narrowing algorithms for a network of constraints. BNR-Prolog [51] and its sequel CLP(BNR) [9] provide relational interval arithmetic in a way that is loosely based on Cleary's pre-publication idea, differing somewhat in particulars. Sidebottom and Havens [54] design and implement a version of relational interval arithmetic in the constraint reasoning system Echidna [26]. Based on hierarchical consistency techniques [43], Echidna can handle unions of disjoint intervals. Lhomme [40] analyzes the complexity of consistency techniques for numeric CSP's and proposes partial consistency techniques, whose complexities can be tuned by adjusting the bound width of the resulting intervals. Lee and van Emden [38, 39] generalize Cleary's algorithms for narrowing intervals constrained by any relations p on $I(\mathbb{R})^n$. They also show how the generalized algorithm can be incorporated in CLP($\mathcal{R}$) [32] and CHIP [19] in such a way that the languages' logical semantics is preserved. Lee and Lee [37] propose an integration of constraint interval arithmetic into logic programming at the Warren Abstract Machine (WAM) [3] level. Benhamou *et al* [8] replaces the usual interval narrowing operator of previous interval CLP languages by an operator based on interval Newton method to speed up non-linear constraint solving.

1.2 Organizations of the Dissertation

The thesis is organized as follows. In chapter 2, we provide the theoretical background to this thesis. We outline the concepts of relational interval arithmetic, followed by a description of interval narrowing and interval splitting. The operational semantics of ICLP($\mathcal{R}$) [38], which is shared by our proposed interval constraint logic programming system, is also presented. In chapter 3, we give a detailed discussion of the limitations of interval narrowing and interval splitting. We verify our assertion empirically by running experiments on some interval narrowing based systems. In chapter 4, we show how to extend ICLP($\mathcal{R}$) with an efficient linear constraint solver, resulting in a new interval CLP system, CIAL (for *Constraint Interval Arithmetic Language*). The architecture of CIAL and the interaction among modules are explained. In chapter 5, we give two proposals for a sound and efficient linear solver. They are based on adaptation of the Gaussian elimination procedures and the Gauss-Seidel method to incremental interval constraint solving respectively. The soundness of the two proposed solvers are also established. In chapter 6, we describe several prototype implementations of CIAL and compare them to other major interval CLP systems. In chapter 7, we summarize our contributions and shed light on further work.

1.3 Notations

This thesis involves several kinds of variables, including *logical variables* (in CIAL) and *mathematical variables* (in algebra), which are either in interval or real domains. To facilitate subsequent discussions, we fix some notations.

Constraints in CIAL are over real numbers. An interval is represented by an appropriate pair of inequality constraints bounding the value of a logical variable, which represents an unknown real number.

[1] Prolog III provides the option of using floating-point arithmetic, although the default is rational arithmetic.

We denote logical variables by such typewriter-like upper case letters as $\mathtt{X}, \mathtt{Y}$ and $\mathtt{Z}$. For example, $\{\mathtt{X} > 3, \mathtt{X} \leq 6\}$ denotes the relation $\mathtt{X} \in (3, 6]$. Mathematical interval variables (or constants) referring to non-empty floating-point intervals are denoted by upper (or lower) case letters with superscript I, while real variables (or constants) are denoted by ordinary upper (or lower) case letters.

Upper case letters in boldface denote matrices, e.g. $\mathbf{A} = (a_{ij}), \mathbf{B}^{\mathbf{I}} = (b_{ij}^I)$, etc. Column vectors are denoted by arrowed letters, such as $\vec{X} = (X_1, \ldots, X_n)^T$, where the superscript T indicates the transpose of a matrix. We overload the $\sum$ symbol to denote summation in the real, floating-point, and interval domains. The exact meaning of the symbol can be inferred from the context of where the symbol appears.

Chapter 2

Overview of ICLP($\mathcal{R}$)

This chapter provides the theoretical background to this thesis. The basics of interval arithmetic, in both functional and relational forms, are presented. We then describe the syntax and semantics of the ICLP($\mathcal{R}$) language [38], which is an extension of CLP($\mathcal{R}$) with relational interval arithmetic. Most of the above materials are adopted from [38, 36, 39] except those otherwise specified.

2.1 Basics of Interval Arithmetic

The manuscripts [45, 4] provide good introduction to interval analysis. Let $\mathbb{R}$ be the set of real numbers and $\mathbb{F}$ the set of floating-point numbers. Mathematically, a *real interval* is a segment, possibly infinite, of the real line and can be defined by an ordered pair of real numbers $a \leq b$, where a is the lower bound and b is the upper bound. For those intervals without upper bound or lower bound, we use the symbols $-\infty$ and $+\infty$ as bounds respectively. Note that $-\infty$ and $+\infty$ can only be used with open bounds. An interval is represented by the usual mathematical notation, such as $[1, 10)$ which denotes the set $\{x \mid 1 \leq x < 10\}$. We differentiate between real intervals and floating-point intervals. The bounds of the elements of the former are real numbers; while the bounds of elements of the latter are restricted to floating-point numbers.

The *set of real intervals*, $I(\mathbb{R})$, is defined by,

$$\begin{aligned} I(\mathbb{R}) \quad = \quad & \{(a,b] \mid a \in \mathbb{R} \cup \{-\infty\}, b \in \mathbb{R}\} \cup \{[a,b) \mid a \in \mathbb{R}, b \in \mathbb{R} \cup \{+\infty\}\} \cup \\ & \{[a,b] \mid a,b \in \mathbb{R}\} \cup \{(a,b) \mid a \in \mathbb{R} \cup \{-\infty\}, b \in \mathbb{R} \cup \{+\infty\}\}. \end{aligned}$$

The set of floating-point intervals is denoted by $I(\mathbb{F})$. We can verify that $I(\mathbb{F}) \subset I(\mathbb{R})$.

If $\cdot \in \{+, -, \times, /\}$, the corresponding *floating-point interval operations* are denoted by,

$$A^I \odot B^I = \{a \cdot b \mid a \in A^I, b \in B^I\}.$$

In the case of interval division, $\oslash$, we assume that B^I does not contain 0.

The basic idea of interval arithmetic is to compute inclusions of arithmetic functions of intervals, and guarantee that the interval outputs will always include all the real solutions. When realizing interval computation on a computer, care must be taken since only a finite subset of real numbers can be exactly represented. Floating-point interval is not closed under the basic interval arithmetic operators, $\oplus, \ominus, \otimes$ and $\oslash$. To preserve the inclusion property, rounded interval arithmetic [45], which is a modification of exact interval arithmetic, is introduced.

If an endpoint of an interval is not a member of $\mathbb{F}$, rounding is made. The rounded floating-point interval is always wider than the original one. Since rounding often occurs in machine computation, we must keep the rounded interval as close to its original real interval as possible. We round the non-representable endpoint to the adjacent floating-point number (round towards $+\infty$ for upper bound and towards $-\infty$ for lower bound). This operation can be formally expressed by the *outward-rounding function*, $\xi : I(\mathbb{R}) \rightarrow I(\mathbb{F})$. If J^I is a non-empty real interval,

$$\xi(J^I) = \bigcap\{J^{I'} \in I(\mathbb{F}) \mid J^I \subseteq J^{I'}\}.$$

The outward-rounding function gives the tightest floating-point interval containing J^I.

2.2 Relational Interval Arithmetic

Cleary [16] introduces "logical arithmetic" by defining distinct primitive arithmetic constraints over intervals, which remove the values of intervals that do not satisfy the constraints. Lee [36] generalizes Cleary's algorithms to *interval reduction*, which is applicable to any arithmetic relation p on $I(\mathbb{R})^n$. Interval reduction can only work on a single constraint. In practice, several constraints interact with one another in a system. A relaxation algorithm is designed to coordinate the application of interval reduction on a set of interval constraints.

2.2.1 Interval Reduction

An *interval constraint* is of the form $(p, \vec{I^I})$, where p is a relation on $\mathbb{R}^n$ and $\vec{I^I} =< I_1^I, I_2^I, \cdots, I_n^I >$ is a tuple of floating-point intervals. The constraint enforces that

$$\exists X_i \in I_i^I \text{ such that } p(X_1, X_2, \cdots, X_n) \text{ holds.}$$

Given the above interval constraint, we try to eliminate the values of each variable that do not satisfy the constraint. Interval reduction effects such an infeasible value elimination.

Interval reduction is defined as an input-output pair. We associate the input n-ary constraint n set-valued functions:

$$\begin{aligned} & F_i(p)(S_1^I, \ldots, S_{i-1}^I, S_{i+1}^I, \ldots, S_n^I) \\ = \; & \pi_i((S_1^I \times \cdots \times S_{i-1}^I \times \pi_i(p) \times S_{i+1}^I \times \cdots \times S_n^I) \bigcap p), \end{aligned}$$

where $i = 1, \ldots, n$, the S_i^I's are intervals, and π_i is the projection function defined by

$$\pi_i(p) = \{s_i | (s_1, \ldots, s_n) \in p\}.$$

Each function gives the set of possible values for the i-th argument of p if the values of all other arguments are restricted to $S_1^I, \cdots, S_{i-1}^I, S_{i+1}^I, \cdots, S_n^I$. The output of interval reduction is defined as,

$$\vec{I^I}' =< I_1^{I'}, \ldots, I_n^{I'} >, \quad \text{where} \quad I_i^{I'} = I_i^I \bigcap \xi(F_i(p)(I_1^I, \cdots, I_{i-1}^I, I_{i+1}^I, \cdots, I_n^I)).$$

If any of the $I_i^{I'}$ is empty, interval reduction *fails* and we can conclude that the constraint $(p, \vec{I^I})$ is *inconsistent*.

Geometrically, interval reduction can be interpreted as intersecting the relation p with the Cartesian product of I_i^I, and then projecting the result onto each coordinate axis. Figure 2.1 shows interval reduction on a simple constraint $(p, < I^I, J^I >)$. The initial floating-point intervals are I^I and J^I. The straight-lined region denotes $I^I \times J^I$ and the shaded region denotes the relation p. The $I^{I'}$ and $J^{I'}$ are the resultant floating-point intervals after the application of interval reduction.

2.2.2 Arithmetic Primitives

A relational interval arithmetic system must support some primitive relational arithmetic operators, such as addition, multiplication and inequalities. The other complicated constraints can be built from these primitives[1].

[1] Users can write the constraints in more convenient notation, such as "$X * Y + 3 * Z > 5 * Y$," but they are eventually translated to conjunctions of primitive constraints, possibly with the aid of extra variables.

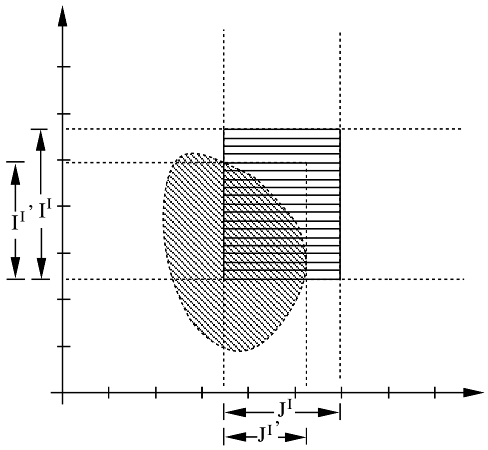

Figure 2.1: Interval Reduction on a Constraint $(p, < I^I, J^I >)$

To have a primitive operator on a relation p, two conditions must hold. First, we need to know how to calculate each function $F_i(p)$ associated with the relation p. Second, in interval arithmetic, we should guarantee intervals are closed under all the defined operators. Therefore, the functions $F_i(p)$ must map from intervals to intervals. A relation which does not satisfy the above conditions can be decomposed into a set of simplier relations.

We only present the primitive relations which are essential to this thesis. Two of them, $\texttt{linear}_\texttt{n}$ and `square`, are newly defined for the implementation of our constraint interval arithmetic language.

Inequalities

$$\texttt{le} = \{(x, y) \mid x, y \in \mathbb{R}, x \leq y\} \qquad \texttt{lt} = \{(x, y) \mid x, y \in \mathbb{R}, x < y\}$$

The functions $F_1(\texttt{le})$ and $F_2(\texttt{le})$ are:

$$\begin{aligned} F_1(\texttt{le})(I_2^I) &= \begin{cases} (-\infty, b] & \text{if } I_2^I = (a, b] \text{ or } [a, b] \\ (-\infty, b) & \text{if } I_2^I = (a, b) \text{ or } [a, b) \end{cases} \\ F_2(\texttt{le})(I_1^I) &= \begin{cases} [a, +\infty) & \text{if } I_1^I = [a, b] \text{ or } [a, b) \\ (a, +\infty) & \text{if } I_1^I = (a, b] \text{ or } (a, b). \end{cases} \end{aligned}$$

Similarly, the functions $F_1(\texttt{lt})$ and $F_2(\texttt{lt})$ are:

$$\begin{aligned} F_1(\texttt{lt})(I_2^I) &= (-\infty, b) && \text{if } I_2^I = [a, b] \text{ or } [a, b) \text{ or } (a, b] \text{ or } (a, b) \\ F_2(\texttt{lt})(I_1^I) &= (a, +\infty) && \text{if } I_1^I = [a, b] \text{ or } [a, b) \text{ or } (a, b] \text{ or } (a, b). \end{aligned}$$

It is trivial to see that $F_i(\texttt{le})$ and $F_i(\texttt{lt})$ map from real intervals to real intervals.

Linear Equality

Cleary [16] proposes a 3-ary `add` primitive,

$$\texttt{add} = \{(x, y, z) \mid x, y, z \in \mathbb{R}, x + y = z\}.$$

This design is too restrictive and may cause unnecessary decompositions of constraints. We give a more general relation $\texttt{linear}_\texttt{n}$ here. For $n \geq 2$,

$$\begin{aligned}\texttt{linear}_\texttt{n} = \{(x_1, x_2, \cdots, x_n) \mid \quad & x_1, x_2, \cdots, x_n, c_1, c_2, \cdots, c_n, c_{n+1} \in I\!R, \\ & c_1 x_1 + c_2 x_2 + \cdots + c_n x_n = c_{n+1}\}.\end{aligned}$$

The function $F_i(\texttt{linear}_\texttt{n})(I_1^I, \cdots, I_{i-1}^I, I_{i+1}^I, \cdots, I_n^I)$ is:

$$F_i(\texttt{linear}_\texttt{n})(I_1^I, \cdots, I_{i-1}^I, I_{i+1}^I, \cdots, I_n^I) = (\xi(c_{n+1}) \ominus \sum_{j=1, j \neq i}^{n} (\xi(c_j) \otimes I_j^I)) \oslash \xi(c_i). \tag{2.1}$$

To show that the function $F_i(\texttt{linear}_\texttt{n})$ always gives real intervals, we only need to consider the $\oslash$ part of definition (2.1) since real intervals are closed under $\oplus$, $\ominus$ and $\otimes$. In definition (2.1), c_i is a real number. We can verify easily that $0 \notin \xi(c_i)$ and $A^I \oslash B^I \in I(I\!R)$ if $0 \notin B^I$.

Multiplication

$$\texttt{mult} = \{(x, y, z) \mid x, y, z \in I\!R, xy = z\}$$

The $\texttt{mult}$ constraint is complicated since it involves both interval multiplication and division. The result of division can be a union of two disjoint intervals in general. This does not satisfy the criterion that the function of primitive relation must map from real intervals to real intervals.

Cleary [16] suggests to decompose $\texttt{mult}$ into $\texttt{mult}^+$ and $\texttt{mult}^-$, where

$$\begin{aligned}\texttt{mult}^+ &= \{(x, y, z) \mid x, y, z \in \Re, x \geq 0, xy = z\} \\ \texttt{mult}^- &= \{(x, y, z) \mid x, y, z \in \Re, x < 0, xy = z\}.\end{aligned}$$

We perform interval reduction on one partition and the another one is visited upon backtracking or under user control.

The definitions of the functions $F_i(\texttt{mult}^+)$ and $F_i(\texttt{mult}^-)$ [39] are,

$$\begin{aligned}F_1(\texttt{mult}^+)(I_2^I, I_3^I) &= (I_3^I \oslash I_2^I) \bigcap I\!R^+ \\ F_2(\texttt{mult}^+)(I_1^I, I_3^I) &= I_3^I \oslash (I_1^I \bigcap I\!R^+) \\ F_3(\texttt{mult}^+)(I_1^I, I_2^I) &= (I_1^I \bigcap I\!R^+) \otimes I_2^I \\ \\ F_1(\texttt{mult}^-)(I_2^I, I_3^I) &= (I_3^I \oslash I_2^I) \bigcap I\!R^- \\ F_2(\texttt{mult}^-)(I_1^I, I_3^I) &= I_3^I \oslash (I_1^I \bigcap I\!R^-) \\ F_3(\texttt{mult}^-)(I_1^I, I_2^I) &= (I_1^I \bigcap I\!R^-) \otimes I_2^I.\end{aligned}$$

The $I\!R^+$ and $I\!R^-$ in the above equations denote the non-negative and negative part of $I\!R$ representatively.

Real interval is shown to be closed under both the $F_i(\texttt{mult}^+)$ and $F_i(\texttt{mult}^-)$.

Square

It is well-known that interval arithmetic suffers from the *variable dependency problem* [22] that causes it to produce inaccurate results. When a given variable occurs more than once in an interval computation, it is treated as a different variable in each occurrence. This causes widening of the computed intervals. A simple example is the fact that if $X^I = [-a, a]$,

$$X^I \otimes X^I = [-a^2, a^2] \neq [0, a^2] \qquad \text{unless } a = 0.$$

The effect of variable dependency problem cannot be eliminated in general. However, it can be alleviated if we can recognize the variable identities in some simple cases. We introduce the relation `square`

$$\texttt{square} \;= \{(x,y) \mid x, y \in \mathbb{R}, x^2 = y\}.$$

The user should specify $(\texttt{square}, < X, Y >)$ instead of $(\texttt{mult}, < X, X, Y >)$ to get a sharp result. A runtime optimization can also be implemented. When a `mult` constraint is encountered, we check the instantiation pattern of variables and determine if the `mult` should be replaced by a `square`.

Similar to `mult`, the `square` relation does not map from real intervals to real intervals. We partition it into $\texttt{square}^+$ and $\texttt{square}^-$, where,

$$\begin{aligned}\texttt{square}^+ &= \{(x,y) \mid x, y \in \mathbb{R}, x \geq 0, x^2 = y\}\\ \texttt{square}^- &= \{(x,y) \mid x, y \in \mathbb{R}, x < 0, x^2 = y\}.\end{aligned}$$

The functions $F_1(\texttt{square}^+)$ and $F_2(\texttt{square}^+)$ are:

$$\begin{aligned}F_1(\texttt{square}^+)(I_2^I) &= (\textcircled{s}(I_2^I \bigcap \mathbb{R}^+))\\ F_2(\texttt{square}^+)(I_1^I) &= (I_1^I \otimes I_1^I) \bigcap \mathbb{R}^+,\end{aligned}$$

where

$$\textcircled{s}(I^I) = \{+\sqrt{a} \mid a \in I^I\}.$$

Similarly, we have for the relation $\texttt{square}^-$:

$$\begin{aligned}F_1(\texttt{square}^-)(I_2^I) &= [0,0] \ominus (\textcircled{s}(I_2^I \bigcap \mathbb{R}^+))\\ F_2(\texttt{square}^-)(I_1^I) &= (I_1^I \otimes I_1^I) \bigcap \mathbb{R}^+\end{aligned}$$

It is obvious that the real intervals are closed under the $\textcircled{s}$ operator. It follows that the functions $F_i(\texttt{square}^+)$ and $F_i(\texttt{square}^-)$ map from real intervals to real intervals.

2.2.3 Interval Narrowing and Interval Splitting

Interval reduction only applies to individual constraint. In practice, there are usually more than one constraint in a relational interval arithmetic system, resulting in a *constraint network*. The constraints will interact with one another by sharing intervals. We need an algorithm to coordinate the execution of interval reduction to narrow the interval constraints in a constraint network.

An interval constraint $(p, \vec{I^I})$ is *stable* if applying interval reduction on $\vec{I^I}$ results in $\vec{I^I}$, otherwise it is *active*. A network is *stable* if all the constraints inside are stable. A *relaxation algorithm* (Algorithm 2.1) reduces a network into a stable one. The relaxation algorithm is similar to the arc-consistency algorithm AC-3 [42]. The use of interval reduction to narrow interval constraints is not mandatory, but can be replaced by any appropriate domain restriction operator [36]. We refer to the relaxation algorithm with interval reduction operator as *interval narrowing*[2].

Interval narrowing sometimes fails to narrow intervals to useful widths. Interval splitting is a divide-and-conquer algorithm used for obtaining sharp solution intervals. An interval is *narrowable* if its width is larger than or equal to a user-defined value. Upon invocation of splitting on a narrowable interval, the interval is partitioned into two halves[3]. We first visit one half, while the remaining half is visited upon backtracking or under user-control. The procedure of interval narrowing with splitting is shown as algorithm 2.2.

[2] Different definitions are used in [36, 9]. In [36], interval narrowing refers to the interval reduction operator described here, while relaxation algorithm is assumed to have interval reduction as the domain restriction operator all the time. Interval reduction and interval narrowing described here are named as narrowing function and narrowing algorithm respectively in [9].

[3] An interval can be partitioned in different ways. Cleary [16] discusses two predicates, `linear_split/1` and `exp_split/1`, which partition intervals at different points. In the implementation of our language, we always partition an interval at its mid-point.

```
let A be an active list which contains active constraints
let P be a passive list which contains stable constraints

while A is not empty
    remove a constraint (p, I⃗^I) from A
    apply interval reduction on (p, I⃗^I) to obtain I⃗^I'
    if interval reduction fails then
        exit with failure
    else
        if I⃗^I ≠ I⃗^I' then
            replace I⃗^I by I⃗^I'
            foreach constraint (q, J⃗^I) in P
                if I⃗^I and J⃗^I share narrowed variable(s) then
                    remove (q, J⃗^I) from P and append it to A
                endif
            endforeach
        endif
    endif
    append (p, I⃗^I) to the end of P
endwhile
```

Algorithm 2.1: Relaxation Algorithm

2.3 Syntax and Semantics

ICLP($\mathcal{R}$) and CLP($\mathcal{R}$) share the same syntax and declarative semantics [31, 32]. An interval constraint in ICLP($\mathcal{R}$) is expressed as,

$$X_1 \in I_1^I, \cdots, X_n \in I_n^I,\ p(X_1, \ldots, X_n),$$

where $X_i \in I_i^I$ is an appropriate pair of inequalities. The operational semantics is based on the generalized $\mathcal{M}_\chi$ derivation [36], which is shown as the following.

Let P be an interval CLP program and G_0 be a goal in the form $\vec{\Sigma}$?– $\vec{\Theta}, \vec{\Delta}$, where $\vec{\Sigma}$ is a set of stable constraints, $\vec{\Theta}$ is a set of active constraints and $\vec{\Delta}$ is a set of atoms. Initially $\vec{\Sigma}$ is empty. A derivation step that reduces a goal G_p to another G_{p+1} follows:

- $\gamma \in \vec{\Delta}$ and the program P contains a rule $R, H :- \vec{\Theta}', \vec{\Delta}'$, that the head atom H can be unified with γ, i.e. $H\theta = \gamma\theta'$. G' is

$$(\vec{\Sigma} \bigcup \vec{\Theta})\theta' \ ?\!-\ \vec{\Theta}'\theta, (\vec{\Delta} \setminus \gamma)\theta' \bigcup \vec{\Delta}'\theta.$$

- G_{p+1} is the sequence of G' with the set of constraints $(\vec{\Sigma} \bigcup \vec{\Theta})\theta'$ replaced by $F_{nf}((\vec{\Sigma} \bigcup \vec{\Theta})\theta')$, where F_{nf} is a *normal function* that maps from set of constraints to set of constraints in such a way that

$$P \models_{\mathcal{M}_\chi} \exists((\vec{\Sigma} \bigcup \vec{\Theta})\theta') \Leftrightarrow P \models_{\mathcal{M}_\chi} \exists(F_{nf}((\vec{\Sigma} \bigcup \vec{\Theta})\theta')).$$

let Q be a *split queue* which contains variables that specified to split

`while` Q contains narrowable intervals
    partition the first narrowable interval in Q, say V^I, into V_1^I and V_2^I
    trail V_2^I
    replace V^I by V_1^I
    invoke interval narrowing
    `if` interval narrowing fails `then`
        perform backtracking
    `endif`
`endwhile`
dump solutions
perform backtracking

Algorithm 2.2: Interval Narrowing with Splitting

Theorem 2.3.1 [36]: If C' is obtained from C using interval reduction on p, where C is $X_1 \in I_1, \ldots, X_n \in I_n$, $p(X_1, \ldots, X_n)$ and C' is $X_1 \in I_1^{'}, \ldots, X_n \in I_n^{'}$, $p(X_1, \ldots, X_n)$, then

$$\models_{\mathcal{M}_\chi} \exists(C) \Leftrightarrow \models_{\mathcal{M}_\chi} \exists(C').$$

■

Theorem 2.3.1 shows that interval reduction transforms an interval constraint into another one with the same solution space. Interval narrowing, which performs interval reduction repeatedly on interval constraints in a constraint network, is therefore a normal-form function.

A generalized $\mathcal{M}_\chi$ derivation is a sequence of goals, possibly infinite. A derivation is *successful* if it is finite and the last goal is empty; *finitely-failed* if it is finite but the last goal has one or more atoms. The generalized $\mathcal{M}_\chi$ derivation ends with a *floundered goal* if the last goal has one or more stable constraints. Floundered goal gives "incomplete" solutions and should be interpreted as conditional answers [44]. Suppose a non-empty goal $\leftarrow G_1, \cdots, G_n$ is derived from $\leftarrow G_0$ and θ is a composition of all the substitutions. The clause $(G_0 \leftarrow G_1, \cdots, G_n)\theta$ is a *conditional answer* to the original goal.

Chapter 3

Limitations of Interval Narrowing

Interval narrowing with splitting is a common constraint solving technique used in interval constraint logic programming languages [39, 37, 9, 51]. Our experiments show that, in general, this technique is impractical in solving some classes of problems in terms of both computation time and storage. We try to solve a set of randomly generated systems of linear equations in several interval narrowing based systems. None of them can give useful solutions (with width less than or equal to 1) for dense linear systems (without zero-coefficients) of rank greater than 5. For sparse linear systems (up to 60% of coefficients are zeros), only those of rank less than 11 can be solved. We give a detailed discussion of the limitations of interval narrowing in this chapter.

3.1 Computation Inefficiency

Interval narrowing can be classified as a fixed-point iterative method. Its convergence depends highly on the initial bounds of variables and the form of interval constraints. A detailed analysis and discussion can be found in [48]. We give a geometrical interpretation to illustrate the convergence of interval narrowing on a simple linear system. Figure 3.1 shows interval narrowing on the following system

$$\{X = Y, X = -Y\} \tag{3.1}$$

with initial bounds $X, Y \in [-50, 50]$. Recall that interval narrowing on a constraint $(p, < I_1^I, \ldots, I_n^I >)$ can be defined as the projection of the intersection of the Cartesian product of I_i^I and the relation p. As shown in figure 3.1, the intersections are two diagonals of the initial guess region and they always project onto the initial bounds of variables. Thus, no value can be eliminated. More generally, given the system (3.1) with initial bounds $X \in [-a, b], Y \in [-c, d]$, where $a, b, c, d \geq 0$, interval narrowing can never give solution which is sharper than

$$X, Y \in [-\min(a, b, c, d), \min(a, b, c, d)].$$

Interval splitting is a divide-and-conquer algorithm for obtaining sharper interval solutions but it is expensive to perform. The efficiency of interval narrowing with splitting depends on its interval subdividing method and search strategy. To use some ad-hoc subdividing methods and search strategies, some special constraints may be solved more efficiently [51].

Interval splitting is impractical for 3 main reasons:

- Interval splitting is implemented using a Prolog backtracking-like mechanism. A choice point is created for each splitting. In the worst case, no partition can be rejected in each splitting and all the choice points are accumulated. If we split n variables $x_1, x_2, \cdots, x_n$, each into m partitions, it requires $(m-1)n$ times trailing spaces of all variables. Since we usually split intervals to desirable narrow width, m is large and interval narrowing with splitting is demanding in memory space.

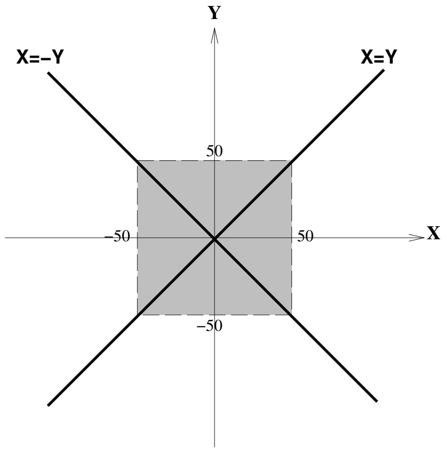

Figure 3.1: Interval Narrowing on $\{X = Y, X = -Y\}$

- We may be unable to obtain sharp results for a system of constraints by invoking splitting on only a variable. Splitting one more variable into d partitions requires d times execution of interval narrowing in the worst case. This increases the computation time rapidly.
- Although special interval subdividing methods and search strategies may improve the efficiency of interval narrowing with splitting, in general, they cannot be known in advance.

We justify our claims using some experimental results. We solve a set of systems of linear constraints,

$$\begin{array}{rl} \mathbf{A}\vec{X} = \vec{b}, \quad \text{where} & \mathbf{A} = (a_{ij}), \vec{X} = (X_i), \vec{b} = (b_i), a_{ij} \neq 0, b_i \neq 0, \\ & \text{and } X_i \in [-10000, 10000] \\ \text{for} & 1 \leq i, j \leq n, \end{array}$$

on BNR Prolog [52], CLP(BNR) [9], Echidna [54] and ICL [37] with splitting. All coefficients a_{ij} and b_i are randomly generated non-zero floating-point numbers. The results for each problem size n are the average of three different sets of test data and are summarized in table 3.1 and table 3.2. The "—" symbol indicates that the test *fails*: either the system halts abruptly (trail/stack overflow) or fails to give solutions with width less than 1. The precision of the answers are set to 10 decimal places for CLP(BNR) and ICL, 5 decimal places for BNR Prolog, and the highest precision (`precision(30)` [55]) for Echidna.

Table 3.1 gives the computation time for problem size ranging from 1 to 6. The computation efficiency of these systems decreases rapidly as n grows. When $n = 4$, Echidna consumes more than 100MB memory and then halts abruptly. CLP(BNR) cannot solve any set of test data with interval narrowing and splitting alone. We have to further apply two predicates `absolve/1` and `presolve/1`, which are designed for solving single non-point solutions and complex problems [50]. Two sets of test data are solved. When $n = 5$, ICL and BNR Prolog can solve only one of the three sets of test data in about 3.5 minutes and 2.6 hours respectively. When $n \geq 6$, all the tested systems either halt abruptly or give wide resultant intervals (with width greater than 1000)[1].

[1] This experiment does not imply that interval narrowing is incapable of solving *any* system of linear constraints with rank greater than 5. Some large systems with special properties, e.g. strictly diagonal dominant, still can be solved even without using interval splitting (see corollary 5.2.6 in chapter 5).

	$n=1$	$n=2$	$n=3$	$n=4$	$n=5$	$n=6$
BNR Prolog‡	6ms	89ms	17.36s	976.28s	9234.07s*	—
CLP(BNR)	10ms	37ms	1.69s	4.80s†	—	—
ICL	11ms	28ms	7.45s	111.77s	210.02s*	—
Echidna§	0s	0s	127s	—	—	—

Table 3.1: Average Computation Time in Solving $\mathbf{A}\vec{X} = \vec{b}$

§Due to the lack of built-in timing predicate, the time is rounded to the nearest second.
‡BNR Prolog runs on a Macintosh II while the other systems run on a SUN SPARCstation 10 with higher precision. BNR Prolog's results should not be directly compared with others. We are interested in its rate of increase, however.
†The result is the average computation time of two sets of test data.
*The result is the computation time for only one set of test data.

	`add`	`mult`$^+$	`mult`$^-$	`splitting`
$n=1$	0	1	1	0
$n=2$	70	50	87	0
$n=3$	3.95×10^4	2.24×10^4	3.95×10^4	44
$n=4$	6.01×10^5	3.75×10^5	5.10×10^5	145
$n=5$	$1.27 \times 10^{6*}$	$6.67 \times 10^{5*}$	$9.99 \times 10^{5*}$	111*
$n=6$	—	—	—	—

Table 3.2: Average Number of Interval Operations in Solving $\mathbf{A}\vec{X} = \vec{b}$ in ICL

*It is the number of interval operations for only one set of test data.

Table 3.2 gives the total of interval operations involved. We only consider the ICL system since we cannot access similar benchmarks for other systems. It is interesting to find that it involves nearly three hundred thousand interval operations in solving a small system ($n = 5$) of constraints.

The experiments described deal with linear systems without zero-coefficients. Linear systems from real-life applications, however, are usually sparse. We introduce sparsity into randomly generated linear systems by fixing a certain percentage of coefficients to be zero. In our experiments, we test systems with respectively 20%, 40%, and 60% zero-coefficients[2]. The randomly generated linear sparse systems are again solved on the previous four interval narrowing based systems. The results, each of which is the average of three different sets of test data, are summarized in table 3.3.

The performance of interval narrowing with splitting has not been improved significantly when 20% to 40% of the coefficients are replaced by zeros randomly. Even for sparse linear systems with 60% zero-coefficients, only systems of rank less than 11 can be solved. On the other hand, interval narrowing (even without splitting) gives sharp solutions efficiently for linear systems with triangular coefficient matrices, which have less than 50% of coefficients are zeros. This behavior shows that the efficiency of interval narrowing depends highly on such properties of the coefficient matrices as the distribution of zero-coefficients, rather than the number of zero-coefficients.

The experiments presented so far are limited to linear constraint solving. Benhamou *et al* [8] give

[2] We exclude the systems with 80% zero-coefficients since such randomly generated linear systems are usually inconsistent.

	20% zeros	40% zeros	60% zeros
$n = 5$	2661.45s	8.62s	0.14s
$n = 6$	—	95.95s	4.70s
$n = 7$	—	3620.35s†	13.76s
$n = 8$	—	—	26.05s
$n = 9$	—	—	1298.21s
$n = 10$	—	—	3451.14s
$n = 11$	—	—	—

(a) BNR Prolog‡

	20% zeros	40% zeros	60% zeros
$n = 5$	1.84s*	1.97s	0.13s
$n = 6$	—	0.64s*	0.44s
$n = 7$	—	—	1.90s
$n = 8$	—	—	1.37s
$n = 9$	—	—	3.21s
$n = 10$	—	—	8.78s*
$n = 11$	—	—	—

(b) CLP(BNR)

	20% zeros	40% zeros	60% zeros
$n = 5$	25.63s†	2.47s	0.01s
$n = 6$	—	16.54s†	0.32s
$n = 7$	—	121.58s*	0.13s*
$n = 8$	—	—	2.27s
$n = 9$	—	—	24.54s
$n = 10$	—	—	9.97s*
$n = 11$	—	—	—

(c) ICL

	20% zeros	40% zeros	60% zeros
$n = 5$	1305s*	718s†	0s
$n = 6$	—	130s*	49s
$n = 7$	—	—	274s
$n = 8$	—	—	1570s*
$n = 9$	—	—	1909s*
$n = 10$	—	—	—
$n = 11$	—	—	—

(d) Echidna§

Table 3.3: Average Time in Solving Sparse $\mathbf{A}\vec{X} = \vec{b}$

§Due to the lack of built-in timing predicate in Echidna, the results are rounded to the nearest seconds.

‡BNR Prolog runs on a Macintosh II while the other systems run on a SUN SPARCstation 10 with higher precision. BNR Prolog's results should not be directly compared with others.

†The result is the average computation time of two sets of test data.

*The result is the computation time for only one set of test data.

Constraint in Narrowing	$A \in$	$B \in$	$D \in$
C_1	$[0, \infty)$	$(-\infty, 0]$	$[1, \infty)$
C_2	$[1, \infty)$	$(-\infty, 0]$	$[1, \infty)$
C_1	$[1, \infty)$	$(-\infty, 0]$	$[2, \infty)$
C_2	$[2, \infty)$	$(-\infty, 0]$	$[2, \infty)$
C_1	$[2, \infty)$	$(-\infty, 0]$	$[3, \infty)$
C_2	$[3, \infty)$	$(-\infty, 0]$	$[3, \infty)$
C_1	$[3, \infty)$	$(-\infty, 0]$	$[4, \infty)$
C_2	$[4, \infty)$	$(-\infty, 0]$	$[4, \infty)$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
C_1	$[1.0 \times 10^{16} - 1, \infty)$	$(-\infty, 0]$	$[1.0 \times 10^{16}, \infty)$
C_2	$[1.0 \times 10^{16}, \infty)$	$(-\infty, 0]$	$[1.0 \times 10^{16}, \infty)$
C_1	$[1.0 \times 10^{16}, \infty)$	$(-\infty, 0]$	$[1.0 \times 10^{16}, \infty)$

Table 3.4: Traces of A, B and D (Without Splitting)

examples on non-linear constraint solving. They show that the growth of number of interval operations involved in non-linear constraint solving in interval narrowing based systems is exponential with respect to the problem size. We conclude that interval narrowing with splitting is inefficient in interval constraint solving.

3.2 Inability to Detect Inconsistency

As stated in section 2.3, answers obtained from interval narrowing should be regarded as conditional. A set of inconsistent constraints can be narrowed to become stable without inconsistency being found. A simple example is,

$$\begin{cases} A & + & 1 = D & \quad (C_1) \\ A & + & B = D & \quad (C_2) \\ A & \in & [0, \infty) & \\ B & \in & (-\infty, 0]. & \end{cases} \tag{3.2}$$

Equations (C_1) and (C_2) imply $B = 1$, which contradicts with the fourth constraint $B \in (-\infty, 0]$. The history of the values of A, B and D after each narrowing step is summarized in table 3.4. The traces show that variable B is never changed during narrowing. Variable A forces the lower bound of variable D to increase towards $+\infty$ in the narrowing of C_1. Variables A and D interchange their roles when constraint C_2 is activated. Since a floating-point number in "`double`" has only 16 significant digits[3], the lower bounds of A and D can never reach the largest floating-point number, but a number near 1.0×10^{16} [4]. Both A and D are narrowed in an extremely slow rate. This explains why such a trivial system takes a long time to stablize.

When interval narrowing stablizes, we invoke interval splitting on the 3 variables. Splitting B accelerates the interval narrowing; while splitting on the variables A and D forces their lower bounds to reach the largest floating-point number. The history of the variables after each interval narrowing with splitting (splitting in the sequence of A, B and D) is shown in table 3.5. The symbols ν, ∞^- and $\infty^=$ denote a negative, the largest and the second largest floating-point number respectively.

[3] We assume that the two bounds of an interval are represented by floating-point numbers in the "`double`" format.
[4] In floating-point arithmetic, $a + 1 = a$ if a is a sufficiently large number, e.g. 1.0×10^{16}.

Constraint in Narrowing	$A \in$	$B \in$	$D \in$
C_1	$[1.0 \times 10^{16}, \infty)$	$(-\infty, 0]$	$[1.0 \times 10^{16}, \infty)$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
C_1	$[\infty^{=}, \infty)$	$(-\infty, 0]$	$[\infty^{=}, \infty)$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
C_1	$[\infty^{-}, \infty)$	$(-\infty, 0]$	$[\infty^{-}, \infty)$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
C_2	$[\infty^{-}, \infty)$	$(-\infty, \nu]$	$[\infty^{-}, \infty)$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
C_2	$[\infty^{-}, \infty)$	$(-\infty, -\infty^{-}]$	$[\infty^{-}, \infty)$

Table 3.5: Traces of A, B and D (With Splitting)

The symbols ν, ∞^{-} and $\infty^{=}$ denote a negative, the largest and the second largest floating-point number respectively.

From the last entry in the traces, we find that interval narrowing ends with $[\infty^{-}, \infty) + (-\infty, -\infty^{-}] = [\infty^{-}, \infty)$. This is due to the fact that, by setting the negative rounding direction, the addition of any number and the largest floating-point number results in the largest floating-point number itself, in the IEEE floating-point standard. Inconsistency cannot be detected even when interval splitting is applied.

In our experiment on system (3.2), BNR Prolog and ICL do not terminate in 90 minutes. CLP(BNR) returns "yes" even when interval splitting is applied since interval narrowing stops after only a few iterations. Echidna returns also "yes" with default precision due to the similar reason and exits abruptly with high precision.

An explanation for this phenomenon is as follows. In system (3.2), the constraint $B = 1$ can only be identified by considering the 2 equations as a whole. However, relaxation algorithm is a local consistency algorithm. Only a constraint is considered in each reduction step. Detection of the inconsistency, however, requires a global view of the constraint system. Therefore, interval narrowing fails to detect the trivial inconsistency of (3.2).

This example exhibits two important shortcomings of interval narrowing. First, interval narrowing is "incomplete" in detecting inconsistency. Second, it may take a long time to stablize on an obviously inconsistent system.

3.3 The Newton Language

Benhamou *et al* [8] show recently an improvement on interval narrowing. The results are implemented in the Newton language. In this section, we outline their work and shows that their improvement applies only to interval non-linear constraint solving, but not to linear constraints. The details of Benhamou *et al*'s experiments and analysis are beyond the scope of this thesis. We refer the readers to [8].

Benhamou *et al* replace the interval reduction operator in interval narrowing by a Newton reduction operator, which is a variant of the interval Newton method. Given an interval constraint $(E \diamond 0, \vec{I^I})$ as described in section 2.2.1, where E is an arithmetic term and $\diamond$ is a relational symbol from $\{=, \geq\}$, the

Newton reduction operator is defined as an input-output pair[5]:

$$\begin{array}{ll}\textbf{Input}: & (E \diamond 0, < I_1^I, \ldots, I_n^I >), \text{ where } I_i^I \in I(\mathbb{F}) \text{ and } E \diamond 0 \subseteq \mathbb{R}^n \\ \textbf{Output}: & \vec{I^I}' =< I_1^{I'}, \ldots, I_n^{I'} >, \text{ where } I_i^{I'} = I_i^I \bigcap \xi(N^*(E_i(X_i), E_i^{'}(X_i), I_i^I)).\end{array}$$

$E_i(X_i)$ is obtained by replacing the variables $X_1, \ldots, X_{i-1}, X_{i+1}, \ldots X_n$ and the real arithmetic operators in E by $I_1^I, \ldots, I_{i-1}^I, I_{i+1}^I, \ldots, I_n^I$ and the corresponding interval arithmetic operators; while $E_i^{'}(X_i)$ is the derivative of $E_i(X_i)$ with respect to X_i. The function N^* is defined as,

$$\begin{array}{rcl} N^*(E_i(X_i), E_i^{'}(X_i), I_i^I) & = & J_k^I \quad (k \geq 1) \\ & \text{where} & J_0^I = I_i^I \\ & & J_{l+1}^I = \xi(\texttt{cpt}(J_l^I)) \ominus E_i(\xi(\texttt{cpt}(J_l^I))) \oslash E_i^{'}(J_l^I) \\ & & J_k^I = J_{k-1}^I, \end{array} \tag{3.3}$$

where `cpt` is a function that gives the mid-point of input interval.

In the following, we show that the Newton reduction operator degenerates to the interval reduction operator when solving interval linear constraints. For a general linear constraint

$$c_1X_1 + c_2X_2 + \ldots + c_nX_n = c_{n+1}, \text{ where } c_1, \ldots, c_{n+1}, X_1, \ldots, X_n \in \mathbb{R},$$

we have

$$E_i(X_i) = \xi(c_i) \otimes X_i \ominus \xi(c_{n+1}) \oplus \sum_{j=1, j\neq i}^{n} (\xi(c_j) \otimes I_j^I).$$

Substituting $E_i(X_i)$ into definition (3.3), we get

$$\begin{array}{rcl} J_{l+1}^I & = & \xi(\texttt{cpt}(J_l^I)) \ominus (\xi(c_i) \otimes \xi(\texttt{cpt}(J_l^I)) \ominus \xi(c_{n+1}) \oplus \displaystyle\sum_{j=1, j\neq i}^{n} (\xi(c_j) \otimes I_j^I)) \oslash \xi(c_i) \\ & \supseteq & \xi(\texttt{cpt}(J_l^I)) \ominus \xi(\texttt{cpt}(J_l^I)) \oplus (\xi(c_{n+1}) \ominus \displaystyle\sum_{j=1, j\neq i}^{n} (\xi(c_j) \otimes I_j^I)) \oslash \xi(c_i) \\ & \supseteq & (\xi(c_{n+1}) \ominus \displaystyle\sum_{j=1, j\neq i}^{n} (\xi(c_j) \otimes I_j^I)) \oslash \xi(c_i). \end{array} \tag{3.4}$$

Since the calculation of J_{l+1}^I in (3.4) is independent of J_i^I, the function N^* can be simplified as

$$N^*(E_i(X_i), E_i^{'}(X_i), I_i^I) \supseteq (\xi(c_{n+1}) \ominus \sum_{j=1, j\neq i}^{n} (\xi(c_j) \otimes I_j^I)) \oslash \xi(c_i). \tag{3.5}$$

The right-hand side of (3.5) is the same as the associated function $F_i(\mathtt{linear_n})$ of relation **`linear`** as stated in section 2.2.2. We expect that the Newton algorithm usually gives wider results than those obtained from interval narrowing due to the variable dependency problem.

[5] In order to have consistent notations throughout this thesis, the notations used here are different from those in [8].

Chapter 4

Design of CIAL

Our work is motivated by the inadequacy of interval narrowing for interval linear constraint solving. We propose to extend ICLP($\mathcal{R}$) with an efficient linear constraint solver, resulting in a new interval constraint logic programming system, CIAL (for *Constraint Interval Arithmetic Language*). The syntax and semantics of CIAL are almost identical to those of ICLP($\mathcal{R}$), except that the relational symbol "=" is replaced by "=:=". In this chapter, we outline the modules of CIAL and explain how they interact. Unification between interval variables and other types of data, and decomposition of interval constraints will also be discussed. The design and implementation of efficient linear solvers will be illustrated in chapter 5.

4.1 The CIAL Architecture

Figure 4.1 gives an overview of the CIAL architecture. The *input* and the *engine* components are adaptation of a Prolog interpreter. Their functions include unification, goal reduction, and delivery of constraints collected at each derivation step to the *solver interface*. The interface in turn decomposes and distributes the constraints to the *linear solver* and the *non-linear solver* accordingly.

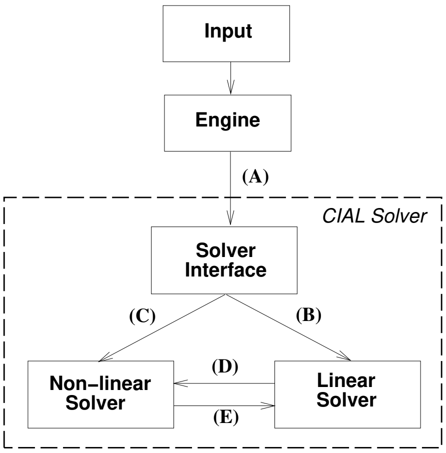

Figure 4.1: An CIAL Architecture

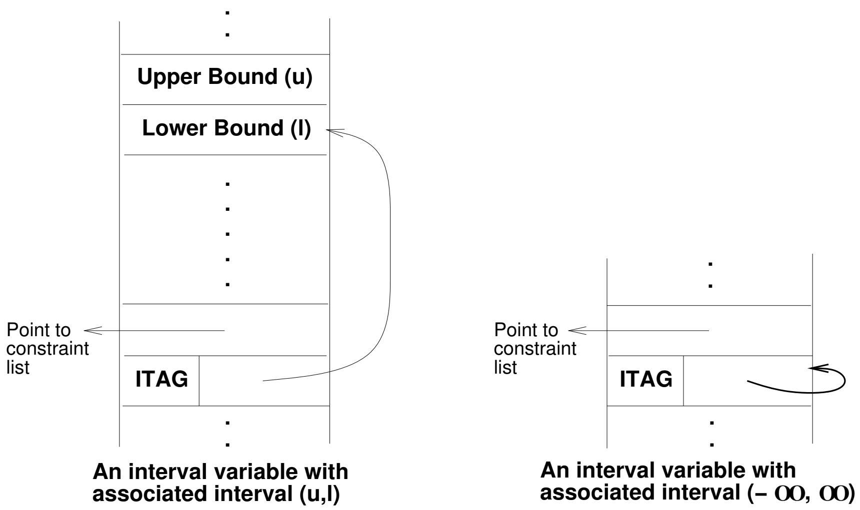

Figure 4.2: The Heap Representations of Interval Variables

In the following, we describe each component of the architecture and the interaction between the two solvers in more details.

4.2 The Inference Engine

The structure of the engine resembles that of a standard structure-sharing Prolog interpreter [3]. Equations between Prolog terms are handled by a standard unification algorithm. Since constraints in CIAL are over *real numbers*, logical variables in constraints denote *unknown real numbers*. We refer to those logical variables as interval variables. The introduction of interval variables calls for an extension of the standard unification algorithm.

4.2.1 Interval Variables

A logical variable `X` becomes an *interval variable* when it is involved in such simple inequality constraints as "$\mathtt{X} > 3, \mathtt{X} \leq 6$," or such equality constraints as "$\mathtt{X} + 2 * \mathtt{Y} =:= \mathtt{Z}, \mathtt{X} * \mathtt{X} =:= \mathtt{Y}$." In the first example, we say that the interval $(3, 6]$ is *associated* with the variable `X`. Semantically, an interval variable is an ordinary logical variable. We distinguish interval variables from logical variables purely for implementation efficiency.

Resembling domain variables in finite-domain languages [57], e.g. CHIP [1, 19], an interval variable is represented as a variable with an associated interval. Its heap representation is shown in figure 4.2, where ITAG is a new tag introduced for interval variables. Each interval variable in CIAL keeps a list of constraints in which the variable appears. When an interval variable is narrowed, we can locate and *wake up* its related constraints efficiently via the constraint list. By waking up a constraint, we mean moving the constraint from the passive list to the active list, This list is important since interval variables are modified often during computation.

4.2.2 Extended Unification Algorithm

Additional binding mechanisms are defined for unification between:

- *an interval variable and a free variable*
  We simply bind the free variable to the interval variable. No constraint solver will be invoked.

- *an interval variable and an interval variable*
  To unify two interval variables `X` and `Y`, we compute the intersection of their associated intervals. If the intersection J^I is non-empty, we choose one of `X` and `Y` (for efficiency reason, we choose the variable which does not require trailing, if possible), say `X`, and bind it to the other, `Y` in this case. Then we replace the associated interval of the chosen variable `X` by the intersection J^I. Otherwise, failure is reported.

- *an interval variable and a number*
  We treat a number as an interval variable, the associated interval of which has the number as closed upper and lower bounds. Thus, unification between a number and an interval variable can be performed in the same way as unification between two interval variables.

- *an interval variable and other terms*
  Failure is reported.

When two interval variables are unified successfully, their constraint lists are merged and all related constraints are waken up.

4.3 The Solver Interface and Constraint Decomposition

The design of the CIAL solver interface is similar to that of CLP($\mathcal{R}$) [32]. The solver interface is called from the inference engine whenever a constraint contains an arithmetic term. If the input constraint contains any number that cannot be represented exactly as a floating-point number, the number will be first outward-rounded to an interval. The constraint is then simplified by evaluating the arithmetic expression. For example, the constraint "`3+9-2*X=:=Y+4*X`" is simplified to "`12=:=Y+6*X`." If the simplified constraint is an equality with only one variable, it is resolved in the interface according to the extended unification algorithm. In all other cases, the input constraint will be decomposed and then distributed to the linear and the non-linear constraint solver accordingly.

CLP($\mathcal{R}$) differentiates between directly solvable constraints and hard constraints [44]. The former is solved by either Gaussian elimination (for linear equalities) or Simplex method (for linear inequalities) once they are collected, while the latter is delayed from consideration until they become linear. We do otherwise in CIAL. We do not delay any constraint. Once constraints are collected in a derivation step, they will be narrowed in either the linear or the non-linear solvers. To maximize the efficiency of constraint solving, we classify constraints into three categories.

An interval is *non-narrowable* if its width is less than a user-defined value or if it cannot be further split in the underlying floating-point system (i.e. when the lower and upper bounds of the interval are "adjacent" in the floating-point line). Otherwise the interval is *narrowable*. A variable is *non-narrowable* if its associated interval is non-narrowable. Otherwise, the variable is *narrowable*. A *constant* is either a floating-point number or a non-narrowable variable. A narrowable variable is a *linear term*. The multiplication of several terms is also a *linear term* if it involves only constants and at most one linear term. Otherwise, the product is a *non-linear term*. A *linear constraint* contains only summation of linear terms and constants, while a *non-linear constraint* contains only summation of non-linear terms and constants. A constraint is *mixed* if it contains both linear and non-linear terms.

In CIAL, linear constraint goes directly to the linear constraint solver without being pre-processed. A non-linear constraint is first partitioned into a set of convex primitives, as described in [16], and then delivered to the non-linear solver. For a mixed constraint, we decompose it into a linear constraint and

1. We *linearize* a mixed constraint by replacing all non-linear terms, each by a temporary variable. Each of the non-linear terms and its corresponding temporary variable are associated by the relational symbol "=:=", resulting in a new non-linear constraint.

2. The linearized constraint is in the form,

$$f(\mathtt{X_1}, \cdots, \mathtt{X_n}) =:= \mathtt{c} + \sum_{k=1}^{j}(s_k \times \mathtt{T_k}),$$

   where $f(\mathtt{X_1}, \cdots, \mathtt{X_n})$ is a linear arithmetic term involving only program/query variables, $\mathtt{X_i}$'s are program/query variables, $\mathtt{c}$ is a constant, $\mathtt{T_k}$'s are the temporary variables introduced to replace non-linear terms, and $s_k = 1$ or -1 for $k = 1, 2, \cdots, j$.

3. The linearized constraint is then partitioned into two by separating the program/query variables from the temporary variables,

$$f(\mathtt{X_1}, \cdots, \mathtt{X_n}) =:= \mathtt{c} + \mathtt{T_0} \quad \text{and} \quad \mathtt{T_0} =:= \sum_{k=1}^{j}(s_k \times \mathtt{T_k}).$$

4. We pass the constraint $f(\mathtt{X_1}, \cdots, \mathtt{X_n}) =:= \mathtt{c} + \mathtt{T_0}$ to the linear solver. The constraint $\mathtt{T_0} =:= \sum_{k=1}^{j}(s_k \times \mathtt{T_k})$ and all non-linear constraints are partitioned into primitives and they are delivered to the non-linear solver.

Algorithm 4.1: Constraint Decomposition Procedure

a set of non-linear constraints. The resultant constraints are handled as ordinary linear or non-linear constraints. The decomposition procedure is shown in algorithm 4.1.

To improve the efficiency of linear constraint solving, we pass the linear constraint $\mathtt{T_0} =:= \sum_{k=1}^{j}(s_k \times \mathtt{T_k})$ to the non-linear solver instead of the linear solver. Interval linear constraint solving usually involves variable elimination [20, 25, 48], which is a time consuming symbolic algorithm. If we deliver the constraint $\mathtt{T_0} =:= \sum_{k=1}^{j}(s_k \times \mathtt{T_k})$ to the linear solver, the temporary variables $\mathtt{T_k}$, for $k = 1, 2, \ldots, j$, are unique there and they can never be eliminated. Thus, such a delivery does not help to give sharper results, but only increases the number of symbolic operations unnecessarily.

We illustrate our constraint decomposition procedure by considering the following query,

$$\begin{array}{ll} ?- & \mathtt{3*X+5*Y-(X+Y)*(6-Z)+Z*Z =:= 10,} \\ & \mathtt{X+Y =:= 20, X*Y*Z =:= 12.} \end{array}$$

The underlined constraints are generated during decomposition.

1. The first constraint is linearized and we have

$$\underline{\mathtt{3*X+5*Y-T1+T2 =:= 10, T1 =:= (X+Y)*(6-Z), T2 =:= Z*Z,}}$$
$$\mathtt{X+Y =:= 20, X*Y*Z =:= 12.}$$

   Note that the term $\mathtt{(X+Y)*(6-Z)}$ should not be further translated into

$$\mathtt{6*X+6*Y-Z*(X+Y)}$$

   which introduces one more occurrence of the variables $\mathtt{X}$ and $\mathtt{Y}$. Such a translation aggravates the effect of the variable dependency problem.

2. We minimize the number of temporary variables in the linearized constraint,

$$\mathtt{3*X+5*Y-T1+T2 =:= 10},$$

by partitioning it into two,

$$\underline{\mathtt{3*X+5*Y+T0 =:= 10, T0 =:= -T1+T2}}, \mathtt{T1 =:= (X+Y)*(6-Z)},$$
$$\mathtt{T2 =:= Z*Z, X+Y =:= 20, X*Y*Z =:= 12}.$$

3. We further decompose the two non-linear constraints,

$$\mathtt{T1 =:= (X+Y)*(6-Z), X*Y*Z =:= 12},$$

into a set of convex primitive constraints.

$$\mathtt{3*X+5*Y+T0 =:= 10, T0 =:= -T1+T2}, \underline{\mathtt{T4 =:= X+Y, T5 =:= 6-Z}},$$
$$\underline{\mathtt{T1 =:= T4*T5}}, \mathtt{T2 =:= Z*Z, X+Y =:= 20}, \underline{\mathtt{X*Y =:= T3, T3*Z =:= 12}}.$$

4. The linear constraints,

$$\mathtt{3*X+5*Y+T0 =:= 10, X+Y =:= 20},$$

are passed to the linear constraint solver, while the others,

$$\mathtt{T0 =:= -T1+T2, T4 =:= X+Y, T5 =:= 6-Z, T1 =:= T4*T5},$$
$$\mathtt{T2 =:= Z*Z, X*Y =:= T3, T3*Z =:= 12},$$

are delivered to the non-linear constraint solver.

4.4 The Linear and the Non-linear Solvers

In traditional interval constraint logic programming languages, all interval constraints are solved under a uniform framework, interval narrowing. To improve the efficiency of interval constraint solving, we separate linear equality constraint solving from inequality and non-linear constraint solving in CIAL.

CIAL consists of two constraint solvers, a linear constraint solver and a non-linear constraint solver. The former is responsible only for linear equality constraints. Non-linear constraints and inequalities belong to the latter. The non-linear constraint solver employs interval narrowing with splitting as the constraint solving technique. The details of constraint solving in the linear solver are discussed in chapter 5. We only outline the main idea here. As described in chapter 3, interval narrowing, which guarantees only local consistency of a system of constraints, is deficient in linear constraint solving. Our linear solver performs global analysis of linear systems by dividing a constraint solving step into two phases. In the first phase, constraints are transformed once they are collected. Global analysis of the linear system is achieved in this transformation. For example, the forward and backward substitutions are such transformations in Gaussian elimination. After all constraints in a derivation step are collected, the associated intervals of variables are narrowed in the second phase by using a domain restriction operator, which maintains local consistency of a single constraint. The application of the domain restriction operations on constraint network is coordinated by relaxation algorithm (algorithm 2.1). This phase is similar to interval narrowing except that a different domain restriction operator from interval reduction is used.

The employment of more than one solver in CIAL calls for an interaction scheme. We explain in algorithm 4.2 how the two solvers cooperate in one constraint solving step. Letters in parentheses refer to the labels in figure 4.1.

Steps 2 and 5-6 correspond to the two phases of the linear constraint solving. Since the transformation method and domain restriction operator depend on the linear constraint solving technique employed, these two steps should be treated as black boxes here. They will be further elaborated in the next chapter.

let LA and NA be two *active lists* which contain active constraints in the linear and the non-linear solvers respectively.

1. A new interval constraint will be resolved in the solver interface if possible. Otherwise, if the constraint is non-linear or mixed, it is decomposed into a conjunction of primitive constraints, or a linear constraint and a conjunction of primitive constraints respectively `(A)`.
2. The linear constraint is sent to the linear solver, transformed, and appended to LA `(B)`.
3. The set of primitive constraints is sent to the non-linear solver and appended to NA `(C)`.
4. After all constraints in a derivation step are collected, the linear constraint solver will be invoked first.
5. Remove a linear constraint from LA and apply a domain restriction operation on it. If any of the variables in the linear constraint is changed, constraints sharing that variable in the linear and the non-linear solvers will be appended to LA and NA respectively `(D)`.
6. Repeat step 5 until LA becomes empty.
7. In the non-linear solver, we apply interval narrowing to make all the constraints there become stable. If a variable is further narrowed and it is also involved in some linear constraints, those constraints will be appended to LA `(E)`.
8. Repeat steps 5 to 7 in a round-robin fashion until both LA and NA become empty.

Algorithm 4.2: Interaction Scheme for Two Solvers

A non-linear primitive constraint is sent to the linear solver only when the constraint becomes linear and it does not contain any temporary variable. Primitive constraints with temporary variables always stay in the non-linear solver since they usually cannot help to eliminate any variable in the constraints in the linear solver[1].

[1] We assume that variable elimination is part of the interval linear constraint solving algorithm.

Chapter 5

The Linear Solver

A good linear solver should satisfy the following criteria:

1. The linear solver must be amenable to efficient *incremental execution*. The complexity of adding and solving a new constraint should be affected more by the form of the new constraint, rather than of the constraints already collected in the linear solver [44].
2. Linear constraint solving in the linear solver must be substantially more *efficient* than interval narrowing.
3. Solutions given by the solver must be *sound* and *accurate*. The former criterion implies that the real solutions should always fall into the answer intervals. To satisfy the latter, the widths of answer intervals should be less than a reasonable value, say 0.001.

In this chapter, we present two proposals [15, 14] to implement such a linear constraint solver. The first, generalized interval Gaussian elimination, is a new combination of CLP($\mathcal{R}$) technology [44] and centered form [45]. This method always yields better results than naive interval Gaussian elimination. The second is a commonly-used iterative interval method, preconditioned interval Gauss-Seidel method [22, 33, 34, 35]. These two methods, as originally designed, operate in the batch mode: all the constraints are collected before solving takes place. In this chapter, we discuss how they can be adapted to incremental execution for use in a CLP system. We conclude this chapter by comparing the two approaches.

5.1 An Interval Gaussian Elimination Solver

Motivated by the success of the linear solver of CLP($\mathcal{R}$), our first proposed linear solver is also an adaptation of the Gaussian elimination procedure. There are several variants of Gaussian elimination procedure for solving linear equalities [58]. An interval version of any of them can be obtained by simply replacing each ordinary arithmetic operator by the corresponding interval arithmetic counterpart. Answers generated using this naive approach, however, will not be as sharp as possible, in general, due to *outward rounding* and *variable dependency problem.* The former is unavoidable in performing interval arithmetic in a floating-point system. Generalized interval arithmetic by Hansen [24] presents a way to reduce the effect of the latter.

5.1.1 Naive Interval Gaussian Elimination

We begin with a quick review of the general form of the Gaussian elimination method in the real number domain. Let $\mathbf{A}$ be an $n \times n$ real matrix and $\vec{b}$ be an n-tuple real vector. To solve

$$\mathbf{A}\vec{X} = \vec{b}, \text{ where } \mathbf{A} = (a_{ij}) \text{ for } i, j = 1, 2, \cdots, n,$$

we perform the Gaussian elimination step (described in (5.1)) $n-1$ times. In the k-th step, we eliminate the elements a_{ik} for $i > k$ by subtracting a suitable multiple of the another row. This procedure, known as *forward substitution*, is effected as,

$$\begin{array}{rcll} b_k^{(k)} & = & b_k^{(k-1)}, & \\ a_{kj}^{(k)} & = & a_{kj}^{(k-1)} & k \le j \le n, \\ a_{ij}^{(k)} & = & a_{ij}^{(k-1)} - a_{kj}^{(k-1)}(a_{ik}^{(k-1)}/a_{kk}^{(k-1)}), & k \le j \le n \text{ and } k+1 \le i \le n \\ b_i^{(k)} & = & b_i^{(k-1)} - b_k^{(k-1)}(a_{ik}^{(k-1)}/a_{kk}^{(k-1)}), & k \le j \le n \text{ and } k+1 \le i \le n \end{array} \quad (5.1)$$

The superscript k denotes the results obtained from the k-th Gaussian elimination step. Eventually, the original equation $\mathbf{A}\vec{X} = \vec{b}$ will be transformed into an upper triangular form, which can be solved by *backward substitution*, defined by

$$X_i \;=\; (b_i - \sum_{j=i+1}^{n} a_{ij}X_j)/a_{ii} \qquad \text{for } i = n, n-1, \ldots, 2, 1.$$

With pivoting [46], this simple algorithm works well in the real domain. If the elements of $\mathbf{A}$ and $\vec{b}$ are intervals, the algorithm performs poorly due to the variable dependency problem.

The variable dependency problem is caused by the fact that multiple occurrences of a given variable in an interval computation are treated as a different variable in each occurrence. This widens the computed interval unnecessarily. As shown in (5.1), the naive Gaussian elimination procedure contains multiple occurrences of almost all coefficients during forward substitution. The simplest examples for illustrating the dependency problem are $A^I \ominus A^I$ and $A^I \oslash A^I$. It can be checked easily that $[0,0] \neq A^I \ominus A^I$ and $[1,1] \neq A^I \oslash A^I$ in general. Consider the following simple system of interval linear equations of two variables:

$$\left\{ \begin{array}{rcl} a_{11}^I \otimes X^I \oplus a_{12}^I \otimes Y^I & = & b_1^I \\ a_{21}^I \otimes X^I \oplus a_{22}^I \otimes Y^I & = & b_2^I. \end{array} \right.$$

Its associated analytical solution is

$$\left\{ \begin{array}{rcl} X^I & = & (b_1^I \otimes a_{22}^I \ominus b_2^I \otimes a_{12}^I) \oslash (a_{11}^I \otimes a_{22}^I \ominus a_{12}^I \otimes a_{21}^I) \\ Y^I & = & (a_{11}^I \otimes b_2^I \ominus a_{21}^I \otimes b_1^I) \oslash (a_{11}^I \otimes a_{22}^I \ominus a_{12}^I \otimes a_{21}^I). \end{array} \right.$$

The variable dependency problem occurs in the calculation of the value of each variable. Substituting the coefficients with example concrete data, we get the following system of equalities.

$$\left\{ \begin{array}{rcl} [3.0002, 3.0003] \otimes X^I \oplus [4.0005, 4.0006] \otimes Y^I & = & [1,1] \\ [2.0001, 2.0002] \otimes X^I \oplus [1.0002, 1.0003] \otimes Y^I & = & [2,2] \end{array} \right. \quad (5.2)$$

Solving the above system using naive interval Gaussian elimination yields the following results.

$$\left\{ \begin{array}{rcl} X^I & = & (1.39966625..., 1.40028177...) \\ Y^I & = & (-0.80016132..., -0.79975469...) \end{array} \right.$$

In the following, we show that better results can be obtained by adopting operators from generalized interval arithmetic in Gaussian elimination.

5.1.2 Generalized Interval Gaussian Elimination

The interval Gaussian elimination procedure is well studied [23, 4, 22, 25]. A well-known algorithm, known as preconditioned interval Gaussian elimination, is proposed by Hansen [23]. Preconditioned Gaussian elimination transforms the coefficient matrix $\mathbf{A^I}$ to a near identity matrix $\tilde{\mathbf{I}}^{\mathbf{I}}$ before applying naive interval Gaussian elimination. Since all subdiagonal elements are nearly zeroes, the variable dependency effect are highly reduced. We do not employ this algorithm in our CIAL linear solver since it is difficult to adapt the algorithm for efficient incremental execution. In Gaussian elimination, an upper triangular matrix is obtained in the symbolic forward substitution phase. Adding an additional constraint to the system in preconditioned interval Gaussian elimination, however, may change all the elements in the coefficient matrix $\tilde{\mathbf{I}}^{\mathbf{I}}$. The upper triangular matrix is difficult to update incrementally if most of the elements in the matrix $\tilde{\mathbf{I}}^{\mathbf{I}}$ are changed, especially in the interval context[1]. Calculating the updated upper triangular matrix from scratch involves re-doing all the symbolic operations. This is time-consuming.

We propose an alternative procedure, generalized interval Gaussian elimination, to reduce the effect of the variable dependency problem in Gaussian elimination. In general, this method cannot give as sharp results as preconditioned interval Gaussian elimination, but always performs better than naive interval Gaussian elimination. The method can also tackle some classes of problems that cannot be handled by interval narrowing based system [15].

A *generalized interval Gaussian elimination* procedure is obtained by replacing ordinary arithmetic operators in an ordinary Gaussian elimination procedure by the corresponding generalized interval arithmetic operators [24], which are described as follows.

In generalized interval arithmetic, an interval $X^I = [a, b]$ is represented as a *generalized interval* of the form,

$$X^I = Y^I \oplus [-c, c] \otimes Z^I \quad \text{where } y - c = a, y + c = b, Y^I = [y, y] \text{ and } Z^I = [1, 1].$$

Suppose $X_i^I = Y_i^I \oplus [-c_i, c_i] \otimes Z_i^I$ for $i = 1, \ldots, n$. If an interval X_{n+1}^I is computed using the n X_i^I's, the resulting interval is also expressed as a generalized interval,

$$X_{n+1}^I = Y_{n+1}^I \oplus \sum_{r=1}^{n} ([-c_r, c_r] \otimes Z_{n+1,r}^I), \text{ [2]}$$

where Y_{n+1}^I and $Z_{n+1,r}^I$'s are numerical intervals computed from the Y^I's and Z^I's of the n input intervals in ordinary interval arithmetic operators. Note that each generalized interval keeps as many subterms as the number of intervals that it depends on. These subterms provide information to locate multiple occurrences of a variable during computation, so as to reduce the effect of the variable dependency problem.

The basic generalized interval arithmetic operators, $\oplus_g$ (addition), $\ominus_g$ (subtraction), $\otimes_g$ (multiplication) and $\oslash_g$ (division), are defined [24] in the following.

Generalized Interval Addition

If $X_k^I = X_i^I \oplus_g X_j^I$, then

$$X_k^I = Y_i^I \oplus Y_j^I \oplus \sum_{r=1}^{n} ([-c_r, c_r] \otimes (Z_{ir}^I \oplus Z_{jr}^I))$$

where

$$Y_k^I = Y_i^I \oplus Y_j^I, Z_{kr}^I = Z_{ir}^I \oplus Z_{jr}^I$$

Generalized Interval Subtraction

If $X_k^I = X_i^I \ominus_g X_j^I$, then

$$X_k^I = Y_i^I \ominus Y_j^I \oplus \sum_{r=1}^{n} ([-c_r, c_r] \otimes (Z_{ir}^I \ominus Z_{jr}^I))$$

[1] The corresponding problem in real domain is usually somewhat easier [21] since we do not need to consider the dependency problem.

[2] We abuse the notation $\sum$ to denote interval summation.

where

$$Y_k^I = Y_i^I \ominus Y_j^I,\, Z_{kr}^I = Z_{ir}^I \ominus Z_{jr}^I$$

Generalized Interval Multiplication

If $X_k^I = X_i^I \otimes_g X_j^I$, then

$$\begin{aligned} X_k^I = Y_i^I \otimes Y_j^I \quad & \oplus \quad \sum_{r=1}^{n}([-c_r, c_r] \otimes (Y_i^I \otimes Z_{jr}^I \oplus Y_j^I \otimes Z_{ir}^I)) \\ & \oplus \quad \sum_{r=1}^{n}\sum_{s=1}^{n}([-c_r c_s, c_r c_s] \otimes Z_{ir}^I \otimes Z_{js}^I) \end{aligned}$$

where

$$\begin{aligned} Y_k^I &= Y_i^I \otimes Y_j^I \oplus \sum_{r=1}^{n}([0, c_r^2] \otimes Z_{ir}^I \otimes Z_{jr}^I) \\ Z_{kr}^I &= Y_i^I \otimes Z_{jr}^I \oplus Y_j^I \otimes Z_{ir}^I + Z_{ir}^I \otimes \sum_{s=1, s\neq r}^{n}([-c_s, c_s] \otimes Z_{js}^I) \end{aligned}$$

Generalized Interval Division

If $X_k^I = X_i^I \oslash_g X_j^I$, then

$$\begin{aligned} X_k^I = Y_i^I \oslash Y_j^I \quad & \oplus \quad (\sum_{r=1}^{n}([-c_r, c_r] \otimes (Y_j^I \otimes Z_{ir}^I \ominus Y_i^I \otimes Z_{jr}^I))) \\ & \oslash \quad (Y_j^I \otimes (Y_j^I \oplus \sum_{s=1}^{n}([-c_s, c_s] \otimes Z_{js}^I))) \end{aligned}$$

where

$$\begin{aligned} Y_k^I &= Y_i^I \oslash Y_j^I \\ Z_{kr}^I &= (Y_j^I \otimes Z_{ir}^I \ominus Y_i^I \otimes Z_{jr}^I) \oslash (Y_j^I \otimes (Y_j^I \oplus \sum_{s=1}^{n}([-c_s, c_s] \otimes Z_{js}^I))) \end{aligned}$$

We give a simple example to show how the effect of the variable dependency problem can be *totally eliminated* (without counting the extremely small errors introduced by outward-rounding) in an expression with *only* generalized interval addition and subtraction. Suppose we want to compute $X_3^I = X_1^I \ominus_g X_2^I \oplus_g X_2^I$ with $X_1^I = [-4, 8]$ and $X_2^I = [6, 8]$. We have

$$\begin{aligned} X_1^I &= [2,2] \oplus [-6,6] \otimes [1,1] \\ X_2^I &= [7,7] \oplus [-1,1] \otimes [1,1] \\ Y_3^I &= Y_1^I \ominus Y_2^I \oplus Y_2^I = [2,2] \ominus [7,7] \oplus [7,7] = [2,2] \\ Z_{31}^I &= Z_{11}^I \ominus Z_{21}^I \oplus Z_{21}^I = [1,1] \ominus [0,0] \oplus [0,0] = [1,1] \\ Z_{32}^I &= Z_{12}^I \ominus Z_{22}^I \oplus Z_{22}^I = [0,0] \ominus [1,1] \oplus [1,1] = [0,0] \\ X_3^I &= Y_3^I \oplus \sum_{s=1}^{2}([-c_s, c_s] \otimes Z_{3s}^I) = [2,2] \oplus [-6,6] \otimes [1,1] = [-4,8]. \end{aligned}$$

Therefore, we get $X_3^I = X_1^I = [-4, 8]$ instead of $[-6, 10]$, which is obtained using ordinary interval arithmetic. Note that if multiplication or division is involved, the dependency problem effect *cannot be eliminated but only reduced.*

Solving the system of linear equation (5.2) using generalized interval Gaussian Elimination yields a sharper result.

$$\begin{cases} X^I &= (1.39981395..., 1.40013404...) \\ Y^I &= (-0.80010553..., -0.79981048...) \end{cases}$$

Implementation Consideration

In our implementation of generalized interval Gaussian elimination, we outward-round the real coefficients in the original (before transformation) system of equations into intervals ($[a_i, b_i]$). Those intervals will then be treated as different *logical variables* with associated bounds represented as generalized intervals,

$$\begin{aligned} \mathtt{X_i} &\in Y_i^I \oplus [-c_i, c_i] \otimes Z_i^I \\ &= [\frac{a_i+b_i}{2}, \frac{a_i+b_i}{2}] \oplus [-\frac{b_i-a_i}{2}, \frac{b_i-a_i}{2}] \otimes [1,1] \end{aligned} \tag{5.3}$$

A generalized interval is in centered form [45] and usually cannot be exactly stored on a computer (i.e. the center point $\frac{a_i+b_i}{2}$ or the quasi-width $\frac{b_i-a_i}{2}$ cannot be exactly represented as a floating-point). We modify equation (5.3) as follows.

$$\begin{aligned} \mathtt{X_i} &\in Y_i^I \oplus (c_i \otimes [-1,1]) \otimes Z_i^I \\ &= [\varphi(\frac{a_i+b_i}{2}), \varphi(\frac{a_i+b_i}{2})] \oplus (\max(\rho(b_i - \varphi(\frac{a_i+b_i}{2})), \rho(\varphi(\frac{a_i+b_i}{2}) - a_i)) \\ &\quad \otimes[-1,1]) \otimes [1,1] \end{aligned} \tag{5.4}$$

In equation (5.4), φ and ρ are functions that round a real number to its nearest and right [49] adjacent floating-point number respectively. We can easily verify that the modified generalized interval obtained from equation (5.4) is a superset of that obtained from equation (5.3). Our experimental results show that using these widened interval coefficients in generalized interval Gaussian elimination still gives sharper solutions than applying naive interval Gaussian elimination to the original linear system.

5.1.3 Incrementality of Generalized Gaussian Elimination

As stated before, incremental execution capability is essential for a good linear solver in constraint logic programming languages. In the following, we present an adaptation of the generalized Gaussian elimination procedure to incremental execution and its interaction with the non-linear solver.

Our algorithm is based on that of CLP($\mathcal{R}$) [32]. Many parts of the CIAL linear solver, such as trailing and backtracking, can be implemented in a similar fashion. We present only the components that differ from their counterparts in CLP($\mathcal{R}$) : *detection of redundancy / inconsistency of newly added linear equation* and the *selection of non-parametric variables.*

All linear equations in the linear solver will be stored in parametric solved form [44] $\mathtt{X} = \sum_{r=1}^{n}(\mathtt{c_r} \times \mathtt{T_r}) + \mathtt{c_{n+1}}$, where $\mathtt{X}$ is a *non-parametric variable*, $\mathtt{T_r}$'s are *parametric variables*, and $\mathtt{c_r}$'s and $\mathtt{c_{n+1}}$ are non-narrowable interval variables that we treat as constants. Assume that we have a collection of consistent linear equations $(E_1, E_2, ...E_{n-1})$ in solved form and a new linear equation E_n is added. Let C_n be the result of substituting out all the non-parametric variables in E_n.

Detection of Inconsistency/Redundancy

Let C_n be in the form $0 = f(\mathtt{T_i})$.

1. If $f(\mathtt{T_i})$ does not contain 0, it implies that the new equation E_n is inconsistent with the stored constraints. Backtracking is needed.

2. If $f(\mathtt{T_i}) = 0$, the new equality is implied by the stored constraints and this new constraint can be removed.

3. A linear constraint is said to be *fixed* if all its parametric variables ($\mathtt{T_i}$) are *non-narrowable*. If C_n is fixed and $f(\mathtt{T_i})$ contains 0, E_n is also redundant. The redundancy of C_n cannot be concluded if any of the variables is narrowable, it is because 0 may be excluded from the value of $f(\mathtt{T_i})$ in the further narrowing of some variables.

Selection of Non-parametric Variables

The candidates of non-parametric variable must not have coefficients containing 0. If there is no variable satisfying this criterion, it implies that C_n cannot be solved by generalized Gaussian elimination. We simply move it to the non-linear solver.

1. If C_n contains new variable(s) that is not currently in the linear solver, choose one of them as the non-parametric variable. This saves the efforts of backward substitution.

2. If C_n contains only parametric variable(s), choose one which appears in the non-linear solver. This criterion is on the contrary to the corresponding rule in CLP($\mathcal{R}$), which tries to select a variable that does not appear in the inequality solver [32]. CIAL hopes that the variables in non-linear constraints can be further narrowed with the aid of linear solver, while CLP($\mathcal{R}$) tries to avoid invocation of the inequality solver.

3. Otherwise, choose the parametric variable with maximum width because it will most probably be narrowed in interval propagation, in general.

Simple Optimizations

In the linear solver, backward substitution is an important step since it can help to eliminate variables and obtain sharper intervals for non-parametric variables. This can in turn re-activate constraints containing the non-parametric variables in the non-linear solver. Backward substitution is, however, also both time and memory consuming. In such cases as $\{\mathtt{X} = \mathtt{Y} + \mathtt{Z}, \mathtt{Y} = \mathtt{A}\}$, the transformation to eliminate Y is fruitless computation. This kind of backward substitution should be delayed. We propose two simple optimizations.

- Backward substitution will not be performed between two constraints if they do not share any common parametric variable.
- A constraint will not be used for backward substitution if it contains an unbound parametric variable $\mathtt{X}$ (i.e. $\mathtt{X} \in (-\infty, \infty)$) and $\mathtt{X}$ does not appear as parametric variable in the constraint being substituted.

5.1.4 Solvers Interaction

In this section, we show how generalized interval Gaussian elimination is augmented with interval narrowing to handle incomplete systems. The enhanced linear constraint solving step interact with the non-linear solver to form a complete constraint solving step.

A linear system may have more unknown variables than equations. Such systems do not have point solutions for each variable even in the real domain. For example, $\{\mathtt{X} > 1, \mathtt{X} < 5, \mathtt{X} + \mathtt{Y} = 20\}$ has solutions $\{\mathtt{X} \in (1,5), \mathtt{Y} \in (15,19)\}$. We call them *incomplete systems.*

In the generalized interval Gaussian elimination procedure, interval propagation proceeds unidirectionally from parametric variables to non-parametric variables instead of being relational as in interval narrowing. This functional propagation works well if there are as many independent equality constraints as the number of variables. In the case of incomplete systems, the extra parametric variables will never be narrowed. *We tackle this problem by combining generalized interval Gaussian elimination and interval narrowing in a linear constraint solving step.* Non-parametric variables are narrowed by interval propagation in interval Gaussian elimination, while parametric variables are narrowed by interval narrowing.

The following steps replace the steps 2 and 5 in algorithm 4.2 to yield a complete constraint solving step. The step 2 performs constraint transformation; while the associated intervals of interval variables are narrowed in the step 5.

2. The linear constraint is sent to the linear solver and transformed by generalized interval Gaussian elimination. If it is either inconsistent or redundant, the solver reports failure or removes the constraint respectively. Otherwise, the new linear constraint and the constraints modified in backward substitutions are appended to the active list LA `(B)`.

5. Remove a linear constraint from LA. The value of the non-parametric variable of the constraint is narrowed by interval propagation from the parametric variables; while the values of its parametric variables are narrowed by interval narrowing. If any of the parametric variables is changed, the constraints in both solvers that share that variable will be appended to LA and NA accordingly. Changing non-parametric variable will never activate other linear constraints but only non-linear constraints `(D)`, since no non-parametric variables can appear as parametric variables in other linear constraints.

We show the soundness of solutions given by the generalized interval Gaussian elimination solver and the termination of the above constraint solving step in the following theorems.

Theorem 5.1.1: Generalized interval Gaussian elimination preserves all solutions of a linear interval system. The solutions given by the generalized interval Gaussian elimination solver are sound.

Proof : From the inclusion monotonicity property of generalized interval arithmetic [24] and the correctness of Gaussian elimination. ■

Theorem 5.1.2: The constraint solving step in algorithm 4.2 with the generalized interval Gaussian elimination solver always terminates.

Proof : Inconsistency can be revealed either in the inconsistency detection phase in generalized interval Gaussian elimination or from empty intervals obtained in the narrowing of interval variables. In both cases, the solvers report failure and the constraint solving step terminates.

Otherwise, the two solvers narrow variables in active constraints (i.e. constraints in active lists), which will be removed from active lists after narrowing. A constraint will be appended to active list in either the cases where the constraint is transformed or some variables in the constraint are narrowed. Constraint transformation will only be performed when some new constraints are added to the system. The number of constraint transformation depends on the number of input constraints, which is always finite. Since the number of floating-point is limited, no variable can be narrowed infinitely. It follows that the constraint solving step must terminate eventually. ■

5.2 An Interval Gauss-Seidel Solver

In many applications, we have some crude bounds on the solution of a linear system $\mathbf{A^I} \otimes \vec{X^I} = \vec{b^I}$. Such a system can be solved efficiently by using some iterative methods. Preconditioned interval Gauss-Seidel method is an iterative method being widely-used in interval computation [48, 22, 33, 34]. We explain how it can be adapted for interval linear constraint solving in CIAL.

5.2.1 Interval Gauss-Seidel Method

Let the i-th equation in $\mathbf{A^I} \otimes \vec{X^I} = \vec{b^I}$ be

$$\sum_{j=1}^{n}(a_{ij}^I \otimes X_j^I) = b_i^I$$

and we have initial bounds on all variables. The interval Gauss-Seidel method works by updating each variable X_i^I by

$$X_i^I \leftarrow ((b_i^I \ominus \sum_{j=1, j\neq i}^{n} (a_{ij}^I \otimes X_j^I)) \oslash a_{ii}^I) \bigcap X_i^I \tag{5.5}$$

in an iterative fashion. If, at any step, any variable becomes the empty interval, then we conclude that the system has no solution.

In an iterative method, a system usually takes more than one iterating cycle to converge. In addition, since we are considering constraint solving in a single processor machine, only one equation can be examined at a time in sequence. The previously computed values can be used as soon as they are available. Assuming that variable updates are coordinated in a naive round-robin fashion, a sequential version of interval Gauss-Seidel is suggested to be [6]

$$X_i^{I(k)} \leftarrow ((b_i^I \ominus \sum_{j=1}^{j=i-1} (a_{ij}^I \otimes X_j^{I(k)}) \ominus \sum_{j=i+1}^{j=n} (a_{ij}^I \otimes X_j^{I(k-1)})) \oslash a_{ii}^I) \bigcap X_i^{I(k-1)} \tag{5.6}$$

The superscript $(k-1)$ of $X_i^{I(k-1)}$ indicates that the variable is obtained in the $(k-1)$-th iterating cycle. The interval Gauss-Seidel method terminates when all variables remain unchanged after an iteration or when the difference between the new and last computed value of each variable is less than a user-defined number. This sequential Gauss-Seidel method is also called the *method of successive displacements* [6].

Convergence

Definition 5.2.1 [48]: A sequence of intervals *converges* iff both the lower and upper bounds converge. ∎

Definition 5.2.2 [48]: The *hull of the solution set* of a linear system is the *set of tightest intervals* that enclosing the solution of the linear system. ∎

In general, interval Gauss-Seidel method cannot be guaranteed to converge to the hull of the solution set of a linear system. We should not expect that it will give sharper results than interval narrowing either since interval Gauss-Seidel method can be considered as "partial" interval narrowing. The following lemmas show these claims.

Definition 5.2.3 [48]: The *magnitude* of an interval $I^I = [l, u]$ is defined as $mag(I^I) = max(|l|, |u|)$, while its *mignitude* is defined as $mig(I^I) = min(|l|, |u|)$, where $|a|$ denotes the *absolute value* of real number a. An interval matrix $\mathbf{A^I} = (a_{ij}^I)$ is said to be *strictly diagonal dominant* if,

$$mig(a_{ii}^I) > \sum_{k=1, k\neq i}^{n} mag(a_{ik}^I) \quad \text{for} \quad i = 1, \ldots, n$$

∎

Lemma 5.2.4 [48]: Interval Gauss-Seidel method is guaranteed to converge to the hull of the solution set[3] of a linear system if the coefficient matrix of the linear system is strictly diagonal dominant. ■

Lemma 5.2.5: Let $\mathbf{A^I} \otimes \vec{X^I} = \vec{b^I}$ be a system of interval linear equalities. If $\vec{X^I}$ and $\vec{X'^I}$ are the solutions obtained from interval narrowing and the interval Gauss-Seidel method respectively, then $\vec{X^I} \subseteq \vec{X'^I}$.

Proof : There exists two differences between the interval Gauss-Seidel method and interval narrowing. First, the interval Gauss-Seidel method considers an interval linear equality as a whole, while interval narrowing decomposes it into a conjuncture of primitives. Constraint decomposition introduces temporary variables. Each temporary variable, say $\mathtt{T}$, occurs twice and appears as in either the form $\{\mathtt{T} = \mathtt{A} \cdot \mathtt{B}, \mathtt{C} = \mathtt{T} \cdot \mathtt{D}\}$ or $\{\mathtt{T} = \mathtt{A} \cdot \mathtt{B}, \mathtt{C} \cdot \mathtt{D} = \mathtt{T}\}$ (The "$\cdot$" symbol denotes $+, -, \times$ or $/$). The variable $\mathtt{T}$ at the left-hand side only serves as a container in interval propagation and it is not involved in any interval computation. No variable dependency effects or rounding-errors will be introduced. This syntactic difference only leads to different narrowing steps or iterating cycles, but never affects the sharpness of the results.

Although both methods use interval propagation to narrow interval variables, interval propagation in the interval Gauss-Seidel method proceeds unidirectionally from subdiagonal variables to diagonal variables instead of being relational as in interval narrowing. It follows that $\vec{X^I} \subseteq \vec{X'^I}$. ■

Corollary 5.2.6: Interval narrowing is guaranteed to converge to the hull of the solution set of a linear system if the original coefficient matrix (before decomposition) of the linear system is strictly diagonal dominant.

Proof : From lemma 5.2.4 and lemma 5.2.5. ■

5.2.2 Preconditioning

As stated in lemma 5.2.4, interval Gauss-Seidel method on a system with strictly diagonal dominant coefficient matrix always converges, but this criterion may not be satisfied in a general system. Hence, one may attempt to transform the system into an equivalent system in the sense that the new system contains all solutions of the original system, but is strictly diagonal dominant. *Preconditioning* effects such a transformation.

Preconditioning is usually done by multiplying a suitable *real* matrix $\mathbf{P}$ to the original system. Instead of solving $\mathbf{A^I} \otimes \vec{X^I} = \vec{b^I}$, we deal with the following system:

$$\mathbf{P} \otimes \mathbf{A^I} \otimes \vec{X^I} = \mathbf{P} \otimes \vec{b^I} \tag{5.7}$$

We call $\mathbf{P}$ the *preconditioner*. Hansen [23] suggests an inverse mid-point matrix as preconditioner which is shown to be optimal [13] in the sense that the preconditioned system gives the tightest bounds of the solutions of the original system. Let $\mathbf{\check{A}}$ denote the *real* mid-point matrix of $\mathbf{A^I}$. We define

$$\begin{array}{lll} \check{a}_{ij} = (l_{ij} + u_{ij})/2 & \text{where} & \mathbf{A^I} = ([l_{ij}, u_{ij}]) \text{ and } \mathbf{\check{A}} = (\check{a}_{ij}) \\ & \text{for} & i, j = 1, 2, \cdots, n. \end{array}$$

We then compute the inverse of $\mathbf{\check{A}}$ using, say row reduction, in high precision floating-point arithmetic. The real $\mathbf{\check{A}^{-1}}$ is used as the preconditioner $\mathbf{P}$ in equation (5.7).

[3] If floating-point interval arithmetic is employed, the solutions obtained are usually slightly wider than the hull of the solution set since outward-rounding is made.

Convergence

Lemma 5.2.7: Let $\mathbf{A^I} \otimes \vec{X^I} = \vec{b^I}$ be a linear system where $\mathbf{A^I}$ is obtained by applying outward-rounding on a real matrix $\mathbf{A}$. The inverse mid-point preconditioned interval Gauss-Seidel method gives solutions which are slightly wider than the hull of the solution set of $\mathbf{A^I} \otimes \vec{X^I} = \vec{b^I}$.

Proof : The inverse mid-point preconditioned interval Gauss-Seidel method can be divided into two phases: performing preconditioning with an inverse mid-point preconditioner and applying interval Gauss-Seidel method on the preconditioned system.

Since all elements in $\mathbf{A^I}$ are obtained by applying outward-rounding on real numbers, their widths should not be wider than the width between two adjacent floating-point numbers. Multiplying such an interval matrix $\mathbf{A^I}$ by its inverse mid-point matrix[4] yields a near identity matrix, which is always strictly diagonal dominant. From lemma 5.2.4, we know that the interval Gauss-Seidel method always converges to the hull of the solution set of the *preconditioned system*. However, due to effect of overestimation (to be discussed later), the preconditioned system usually have slightly wider solutions than the original system. It follows that the inverse mid-point preconditioned interval Gauss-Seidel method gives solutions which are slightly wider than the hull of the solution set of $\mathbf{A^I} \otimes \vec{X^I} = \vec{b^I}$. ■

Intuitively, we can expect that this preconditioned interval Gauss-Seidel method gives accurate results. Since $\mathbf{\check{A}^{-1}} \otimes \mathbf{A^I}$ is near identity, the width of the summation of all sub-diagonal terms $a^I_{ij} \otimes X^I_j$ in equation (5.6) tends to be very small. The diagonal variables are narrowed to be sharp.

Overestimation

Since preconditioning involves many interval multiplications, small errors will be introduced due to outward-rounding. A preconditioned system usually has slightly wider solutions than the original system and these additional pseudo-solutions are called overestimation [47]. Overestimation *destroys the completeness of inconsistency detection* in interval Gauss-Seidel method since an inconsistent system of constraints may become consistent after preconditioning.

A formal analysis of overestimation requires such knowledge as mappings [48] and fixpoint theorems [47], and is beyond the scope of this thesis. Readers may refer to [47, 48] for details.

We end this section by explaining how the simple system $\{X = Y, X = -Y\}$ in section 3.1 can be solved by the preconditioned Gauss-Seidel method without using splitting.

Preconditioning transforms the equations $\{X = Y, X = -Y\}$ (figure 5.1(a)) to $\{X = 0, Y = 0\}$ (figure 5.1(b)), which are the two axes. Even without giving any initial bounds for the two variables, the preconditioned interval Gauss-Seidel method gives the exact answers $\{X = 0, Y = 0\}$ in one iterating cycle. If we apply the preconditioned interval Gauss-Seidel method on general linear systems, initial bounds are usually necessary and it may take several iterating cycles to converge.

5.2.3 Incrementality of Preconditioned Gauss-Seidel Method

The adaptation of the preconditioned interval Gauss-Seidel method for efficient incremental execution is more complicated than that for generalized interval Gaussian elimination. It involves the *incremental update of preconditioner*, *application of preconditioning*, and *detection of inconsistency and redundancy*. A tradeoff between speed and space is also an important consideration in implementing trailing and backtracking. We discuss the stated issues in this section.

[4] The exact inverse mid-point matrix is non-representable, in general. We use an approximation instead.

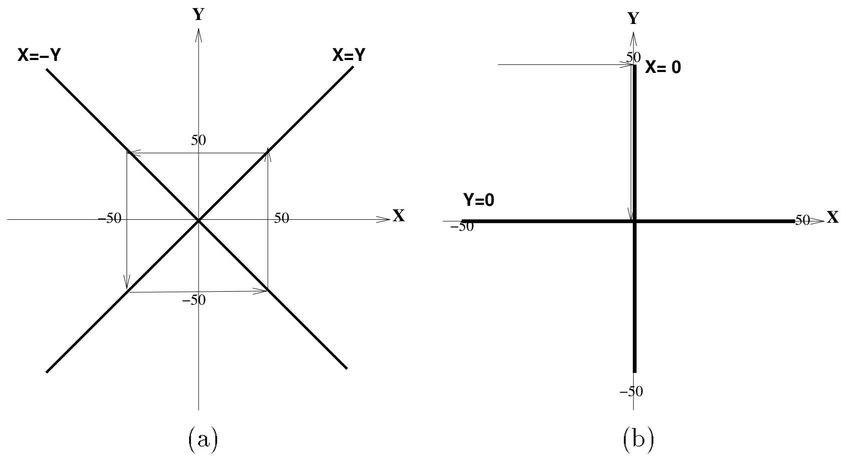

Figure 5.1: Preconditioned Interval Gauss-Seidel Method on Simple Equations

Incremental Update of Preconditioner

We adopt the optimal preconditioner, inverse mid-point matrix as stated in section 5.2.2. Assume that we have a collection of r interval linear equalities of c variables. The mid-point coefficient matrix $\check{\mathbf{A}}$ is thus an $r \times c$ matrix. The entries in $\check{\mathbf{A}}$, which are mid-points of intervals, cannot be represented exactly on computer in general. We can simply round them to their nearest floating-point numbers. The small errors introduced in the mid-point matrix do not affect the convergence of the preconditioned system significantly.

Constraints are generated and submitted to the constraint solver incrementally in a constraint logic programming system. The linear system present in the solver do not necessarily have a square matrix in general. We present how the preconditioner can be computed from such a rectangular matrix.

For the case where $c < r$, it implies that some equalities are either redundant or inconsistent to the system. Those equalities will be located by another algorithm using heuristic (to be discussed later) and they should not be used in the calculation of the inverse. We disregard this case.

Otherwise, we have $c \geq r$. We define a corresponding *rectangular identity matrix* $\mathbf{J}$ by

$$\begin{array}{ll} \mathbf{J} = (j_{kl}), & \text{where} \quad j_{kl} = 1 \text{ for } k = l, \qquad j_{kl} = 0 \text{ for } k \neq l \\ & \text{for} \quad 1 \leq k \leq r \text{ and } 1 \leq l \leq c. \end{array}$$

The preconditioner $\mathbf{P}$ is computed by row reducing the combined matrix $[\check{\mathbf{A}}|\mathbf{J}]$ until the the first r columns of $\check{\mathbf{A}}$ becomes the identity matrix $\mathbf{I}$. The required preconditioner $\mathbf{P}$ resides in the first r columns of the original $\mathbf{J}$ matrix[5]. Therefore, the row reduced matrix has the form $[\mathbf{I}|\mathbf{U}|\mathbf{P}|\mathbf{Z}]$, where $\mathbf{I}$ is the $r \times r$ identity matrix, $\mathbf{U}$ is an $r \times (c-r)$ matrix to be used for future update of the preconditioner, $\mathbf{Z}$ is an $r \times (c-r)$ zero matrix. We call $[\mathbf{I}|\mathbf{U}|\mathbf{P}|\mathbf{Z}]$ the *IUPZ matrix*.

Incremental calculation of the row reduction transformation is achieved by adapting the familiar incremental Gaussian elimination of CLP($\mathcal{R}$) [44]. Assume that we have a collection of r interval linear equalities of c variables with $r < c$. When a new linear equality, whose mid-point coefficients are denoted by $m_{r+1,l}$, is added, the IUPZ matrix, $\mathbf{X} = (x_{kl})$ for $1 \leq k \leq r, 1 \leq l \leq c$, is updated incrementally as in algorithm 5.1.

[5] Note that $\mathbf{P}$ is an $r \times r$ matrix.

1. The IUPZ matrix $\mathbf{X}$ is augmented to $r+1$ rows by appending an extra row where

$$\begin{array}{lcll} x_{r+1,l} & = & m_{r+1,l} & \text{for} \quad 1 \le l \le c \\ x_{r+1,l} & = & 0 & \text{for} \quad c+1 \le l \le 2c \text{ and } l \ne c+r+1 \\ x_{r+1,l} & = & 1 & \text{for} \quad l = c+r+1. \end{array}$$

2. We subtract the $(r+1)$-st row of $\mathbf{X}$ from suitable multiples of the first r rows such that the first r columns of the $(r+1)$-st row becomes zeros.

3. If $c > r+1$, we permute the $(r+1)$-st column of $\mathbf{X}$ with one chosen from the $(r+2)$-nd to c-th columns, say the s-th one. The chosen column must satisfy the following criteria:

   - the component $x_{r+1,s}$ must be greater than a small user-defined value.
   - the column should have as little zeros as possible.

   We also permute the $(c+r+1)$-st with the $(c+s)$-th column accordingly. All permutations are recorded to guarantee that their associated variables in the constraints can be identified.

4. We subtract the first r rows from a suitable multiple of the $(r+1)$-st row such that the $(r+1)$-st columns of the first r rows becomes zeros.

Algorithm 5.1: Incremental Update Procedure for the IUPZ Matrix

The calculation should be performed in high precision floating-point arithmetic. The updated $(r+1) \times (r+1)$ preconditioner resides in the $(c+1)$-st to $(c+r+1)$-st columns of the IUPZ matrix $\mathbf{X}$. We perform permutations in the update procedure for two reasons.

First, since the preconditioner is computed in floating-point arithmetic, to reduce the roundoff errors, we should avoid placing an extremely small floating-point in the diagonal of the matrix. Although roundoff errors introduced in the calculation of preconditioner can never destroy the inclusion property of the preconditioned system, the coefficient matrix of the preconditioned system may not be strictly diagonal dominant. Thus, the preconditioned interval Gauss-Seidel method may fail to converge. This phenomenon becomes more visible if the mid-point coefficient matrix is ill-conditioned [5]. The column permutation in our update algorithm is similar to pivoting in numerical methods. The difference is that the former only requires the diagonal coefficients to be greater than a small user-defined value, while the latter always puts the largest component of a row in the diagonal.

The second criterion, which is to choose a column with as little zeros as possible, is for efficiency only. Recall that when we apply the interval Gauss-Seidel method on a collection of r constraints, only the values of the r variables in the diagonal will be updated. For systems where there are more variables than constraints, extra variables need to be narrowed by interval narrowing, which is time-consuming. A simple way to reduce the number of such extra variables is to include as many non-zero components as possible in the first r columns of the IUPZ matrix in the calculation of the preconditioner.

Detection of Inconsistency/Redundancy

There is no general method to detect redundancy in an iterative method, especially in the interval context. Inconsistency is revealed if an empty intersection is produced by applying the preconditioned Gauss-Seidel method directly to the system. However, overestimation prevents us from detecting all possible inconsistency.

$$P_{((n+1)x(n+1))} \otimes A^{I}_{((n+1)xc)} = \left[\begin{array}{c|c} O_{(nxn)} & L_{(nx1)} \\ \hline M_{(1xn)} & N_{(1x1)} \end{array}\right] \otimes \left[\begin{array}{c|c} A^{I}_{1\,(nxn)} & A^{I}_{2\,(nx(c-n))} \\ \hline A^{I}_{3\,(1xn)} & A^{I}_{4\,(1x(c-n))} \end{array}\right] = \left[\begin{array}{c|c} R^{I}_{1\,(nxn)} & R^{I}_{2\,(nx(c-n))} \\ \hline R^{I}_{3\,(1xn)} & R^{I}_{4\,(1x(c-n))} \end{array}\right]$$

Figure 5.2: Partition of the Preconditioner $\mathbf{P}$ and the Coefficient Matrix $\mathbf{A^I}$

In the implementation of an interval linear constraint solver, we must not use inconsistent or redundant constraints in computing the preconditioner. Intuitively, a system with redundant or inconsistent constraints often have more constraints than the number of unknown variables. Selecting the "wrong" subset of equalities for preconditioning produces a poor preconditioner in the sense that the preconditioned system may fail to converge.

We use a simple heuristic to locate inconsistent and redundant equalities. Incremental calculation of a matrix inverse involves forward substitution (steps 1 and 2 in algorithm 5.1). If we find that after forward substitution on, say, the $(r+1)$-st constraint during the calculation of the inverse of the mid-point coefficient matrix $\breve{\mathbf{A}}$, all of $\breve{\mathbf{A}}$'s remaining $c - r$ coefficient mid-points are less than a small user-defined value, say 10^{-8}, then we conclude that the $(r+1)$-st constraint is either "redundant" or "inconsistent." Inconsistent and redundant constraints are both regarded as *fruitless to preconditioning* and will not be employed in the preconditioning process.

Note that our proposed method is only a heuristics. Constraints concluded to be redundant or inconsistent may indeed be independent and consistent. Simply disregarding these constraints can result in excessively relaxed answer constraints. Current practice in the CIAL system is to transfer these constraints to the non-linear solver for further scrutiny.

Incremental Application of Preconditioning

Preconditioning involves the multiplication of a point preconditioner matrix and an interval matrix, which is of $O(n^3)$ complexity[6], where n is the rank of the preconditioner. Without incremental application, we need to re-compute the preconditioned system from scratch whenever new equalities are added in every derivation step. In the worst case (i.e. when only exactly *one* new equality is collected in each derivation step and the entire preconditioner is modified), the whole constraint solving algorithm has complexity $O(n^4)$. We give an incremental adaptation of preconditioning application of order $O(n^3)$.

Let $\mathbf{O}' \otimes \mathbf{A}'^{\mathbf{I}} \otimes \vec{X}^I = \mathbf{O}' \otimes \vec{b'}^I$ be a preconditioned system with n constraints. Without loss of generality, we assume that one new constraint is added to the system. Our goal is to make use of the previously calculated $\mathbf{O}' \otimes \mathbf{A}'^{\mathbf{I}}$ and $\mathbf{O}' \otimes \vec{b'}^I$ to compute some parts of the new preconditioned system $\mathbf{P} \otimes \mathbf{A^I} \otimes \vec{X}^I = \mathbf{P} \otimes \vec{b}^I$. Note that $\mathbf{A^I}$ differs from $\mathbf{A}'^{\mathbf{I}}$ by only an extra last row.

Consider the computation of $\mathbf{P} \otimes \mathbf{A^I}$. We partition the new preconditioner $\mathbf{P}$, the new coefficient matrix $\mathbf{A^I}$, and their product as shown in figure 5.2. The product matrix $\mathbf{R^I}$ can be calculated by the

[6] We do not consider such special divide-and-conquer matrix multiplication algorithms as Strassen's algorithm ($O(n^{2.81})$) [18]. Those algorithms usually introduce multiple occurrences of variables and require the dimension of the matrix to be a power of 2. The latter can double the storage in the worst case.

following equations:

$$\begin{array}{rcl} \mathbf{R_1^I} & = & \mathbf{O} \otimes \mathbf{A_1^I} \oplus \mathbf{L} \otimes \mathbf{A_3^I} \\ \mathbf{R_2^I} & = & \mathbf{O} \otimes \mathbf{A_2^I} \oplus \mathbf{L} \otimes \mathbf{A_4^I} \\ \mathbf{R_3^I} & = & \mathbf{M} \otimes \mathbf{A_1^I} \oplus \mathbf{N} \otimes \mathbf{A_3^I} \\ \mathbf{R_4^I} & = & \mathbf{M} \otimes \mathbf{A_2^I} \oplus \mathbf{N} \otimes \mathbf{A_4^I} \end{array} \tag{5.8}$$

All terms in (5.8), except $\mathbf{O} \otimes \mathbf{A_1^I}$ and $\mathbf{O} \otimes \mathbf{A_2^I}$, can be calculated in $O(n^2)$ time. We concentrate on the computation of $\mathbf{O} \otimes \mathbf{A_1^I}$ and $\mathbf{O} \otimes \mathbf{A_2^I}$.

The row reduced IUPZ matrix has the form $[\mathbf{I}|\mathbf{U}|\mathbf{O}'|\mathbf{Z}]$ before a new constraint is added. Let $\mathbf{O}' = (o'_{lm})$ for $1 \leq l, m \leq n$ and (u_l) the first column of $\mathbf{U}$. The situation of the IUPZ matrix when a new constraint is added is depicted in (5.9).

$$\left[\begin{array}{cccccc|cccccc} 1 & 0 & \cdots & 0 & u_1 & \cdots & o'_{11} & o'_{12} & \cdots & o'_{1n} & 0 & \cdots \\ 0 & 1 & \cdots & 0 & u_2 & \cdots & o'_{21} & o'_{22} & \cdots & o'_{2n} & 0 & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & \cdots & 1 & u_n & \cdots & o'_{n1} & o'_{n2} & \cdots & o'_{nn} & 0 & \cdots \\ \breve{a}_{n+1,1} & \breve{a}_{n+1,2} & \cdots & \breve{a}_{n+1,n} & \breve{a}_{n+1,n+1} & \cdots & 0 & 0 & \cdots & 0 & 1 & \cdots \end{array}\right] \tag{5.9}$$

To update the preconditioner incrementally, we apply row reduction to the left matrix in (5.9), while the same operations are performed to the right matrix. The first $(n+1)$ columns of the left matrix will be transformed to an identity matrix eventually if the new constraint is independent and consistent to the system. An intermediate state of the row reduction procedure is:

$$\left[\begin{array}{cccccc|cccccc} 1 & 0 & \cdots & 0 & u_1 & \cdots & o'_{11} & o'_{12} & \cdots & o'_{1n} & 0 & \cdots \\ 0 & 1 & \cdots & 0 & u_2 & \cdots & o'_{21} & o'_{22} & \cdots & o'_{2n} & 0 & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & \cdots & 1 & u_n & \cdots & o'_{n1} & o'_{n2} & \cdots & o'_{nn} & 0 & \cdots \\ 0 & 0 & \cdots & 0 & 1 & \cdots & t_1 & t_2 & \cdots & t_n & t_{n+1} & \cdots \end{array}\right]$$

where t_i's are some intermediate values. The updated $(n+1) \times (n+1)$ preconditioner $\mathbf{P}$ is

$$\mathbf{P} = \left[\begin{array}{ccccc} o'_{11} - u_1 t_1 & o'_{12} - u_1 t_2 & \cdots & o'_{1n} - u_1 t_n & -u_1 t_{n+1} \\ o'_{21} - u_2 t_1 & o'_{22} - u_2 t_2 & \cdots & o'_{2n} - u_2 t_n & -u_2 t_{n+1} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ o'_{n1} - u_n t_1 & o'_{n2} - u_n t_2 & \cdots & o'_{nn} - u_n t_n & -u_n t_{n+1} \\ t_1 & t_2 & \cdots & t_n & t_{n+1} \end{array}\right] \tag{5.10}$$

What we have described is the analytic solution of $\mathbf{P}$, which depends by no means on the mode, real or floating-point, of the arithmetic operators. *Assuming that we are using real (interval) arithmetic*, we can establish the following equality and inclusion relationships. We decompose the upper-left $n \times n$ sub-matrix $\mathbf{O}$ of $\mathbf{P}$ as follows.

$$\mathbf{O} = \mathbf{O}' - \left(\begin{array}{c} u_1 \\ u_2 \\ \cdots \\ u_n \end{array}\right) \times (t_1 \; t_2 \; \ldots \; t_n). \tag{5.11}$$

It follows that $\mathbf{O} \otimes \mathbf{A_1^I}$ and $\mathbf{O} \otimes \mathbf{A_2^I}$ can be approximated[7] by

$$\mathbf{O} \otimes \mathbf{A_1^I} \quad \subseteq \mathbf{O}' \otimes \mathbf{A_1^I} \ominus \begin{pmatrix} u_1 \\ u_2 \\ \dots \\ u_n \end{pmatrix} \otimes ((t_1 \ t_2 \ \dots \ t_n) \otimes \mathbf{A_1^I}) \tag{5.12}$$

$$\mathbf{O} \otimes \mathbf{A_2^I} \quad \subseteq \mathbf{O}' \otimes \mathbf{A_2^I} \ominus \begin{pmatrix} u_1 \\ u_2 \\ \dots \\ u_n \end{pmatrix} \otimes ((t_1 \ t_2 \ \dots \ t_n) \otimes \mathbf{A_2^I}). \tag{5.13}$$

using the *subdistributivity*[8] and *associativity*[9] [45] properties of interval arithmetic. *Unfortunately, none of (5.11), (5.12), and (5.13) hold under floating-point (interval) arithmetic* since associativity and subdistributivity are no longer guaranteed.

The right-hand-sides of (5.12) and (5.13) contain $\mathbf{O}' \otimes \mathbf{A_1^I}$ and $\mathbf{O}' \otimes \mathbf{A_2^I}$, slight supersets of which are available from the previous preconditioned system. The multiplication of a vector and a matrix is an $O(n^2)$ operation. Thus the computation of the right-hand-sides of (5.12) and (5.13) is also of $O(n^2)$ complexity. We adapt this more efficient method to precondition the system instead of using $\mathbf{P}$ as defined in (5.10). In the following we state the preconditioning procedure before showing the correctness of the procedure.

The first step is to widen each component of the floating-point vector $(u_1, u_2, \dots, u_n)^T$ by a small amount, say $1e^{-12}$. The result is an interval vector $(u_1^I, u_2^I, \dots, u_n^I)^T$. We define $\mathbf{C_1^I}$ and $\mathbf{C_2^I}$ as follows.

$$\mathbf{C_1^I} = \mathbf{O}' \otimes \mathbf{A_1^I} \ominus \begin{pmatrix} u_1^I \\ u_2^I \\ \dots \\ u_n^I \end{pmatrix} \otimes ((t_1 \ t_2 \ \dots \ t_n) \otimes \mathbf{A_1^I}) \tag{5.14}$$

$$\mathbf{C_2^I} = \mathbf{O}' \otimes \mathbf{A_2^I} \ominus \begin{pmatrix} u_1^I \\ u_2^I \\ \dots \\ u_n^I \end{pmatrix} \otimes ((t_1 \ t_2 \ \dots \ t_n) \otimes \mathbf{A_2^I}) \tag{5.15}$$

We modify the left-hand-side of the preconditioned system by replacing the calculation of $\mathbf{R_1^I}$ and $\mathbf{R_2^I}$. The new calculation is:

$$\begin{array}{rcl} \mathbf{R_1^I} & = & \mathbf{C_1^I} \oplus \mathbf{L} \otimes \mathbf{A_3^I} \\ \mathbf{R_2^I} & = & \mathbf{C_2^I} \oplus \mathbf{L} \otimes \mathbf{A_4^I} \end{array} \tag{5.16}$$

We call the new left-hand-side of the preconditioned system (obtained from (5.8) and (5.16)), $\mathbf{K^I} \otimes \vec{X}^I$. The next step is to find an appropriate floating-point preconditioner $\mathbf{P}'$ to multiply $\vec{b^I}$, the criterion being that $\mathbf{P}' \otimes_r \mathbf{A^I} \subseteq \mathbf{K^I}$, where the symbol $\otimes_r$ denotes the *real* interval multiplication. We propose $\mathbf{P}'$ to be $\mathbf{P}$ with the $\mathbf{O}$ part (as shown in figure 5.2) replaced by $\mathbf{O_{new}}$ defined as follows.

$$\mathbf{O_{new}} \in \mathbf{O}' \ominus_i \begin{pmatrix} u_1^I \\ u_2^I \\ \dots \\ u_n^I \end{pmatrix} \otimes_i (t_1 \ t_2 \ \dots \ t_n), \tag{5.17}$$

[7] A floating-point number a can be regarded as a point interval $[a, a]$. Thus $\mathbf{O}$ can be regarded as a matrix of point intervals.

[8] $A^I \otimes (B^I \oplus C^I) \subseteq A^I \otimes B^I \oplus A^I \otimes C^I$.

[9] $(A^I \otimes B^I) \otimes C^I = A^I \otimes (B^I \otimes C^I)$.

where the symbol $\ominus_i$ and $\otimes_i$ denote the inward-rounded subtraction and multiplication respectively. The definition of *inward-rounding* follows.

Definition 5.2.8: If J^I is a non-empty real interval, the *inward-rounding function* $\eta : I(\mathbb{R}) \to I(\mathbb{F})$ is defined as,

$$\eta(J^I) = \bigcup \{J^{'I} \in I(\mathbb{F}) | J^{'I} \subseteq J^I\}.$$

■

Lemma 5.2.9:

$$\begin{aligned}
\mathbf{O_{new}} \otimes_r \mathbf{A_1^I} &\subseteq \mathbf{O}^{'} \otimes \mathbf{A_1^I} \ominus \begin{pmatrix} u_1^I \\ u_2^I \\ \cdots \\ u_n^I \end{pmatrix} \otimes ((t_1 \ t_2 \ \ldots \ t_n) \otimes \mathbf{A_1^I}) \\
\mathbf{O_{new}} \otimes_r \mathbf{A_2^I} &\subseteq \mathbf{O}^{'} \otimes \mathbf{A_2^I} \ominus \begin{pmatrix} u_1^I \\ u_2^I \\ \cdots \\ u_n^I \end{pmatrix} \otimes ((t_1 \ t_2 \ \ldots \ t_n) \otimes \mathbf{A_2^I})
\end{aligned}$$

Proof : From equation (5.17), we have

$$\mathbf{O_{new}} \in \mathbf{O}^{'} \ominus_i \begin{pmatrix} u_1^I \\ u_2^I \\ \cdots \\ u_n^I \end{pmatrix} \otimes_i (t_1 \ t_2 \ \ldots \ t_n).$$

Let the symbols $\otimes_r$ and $\ominus_r$ denote the *real* interval multiplication and subtraction respectively. It follows,

$$\begin{aligned}
\mathbf{O_{new}} \otimes_r \mathbf{A_1^I} &\subseteq (\mathbf{O}^{'} \ominus_i \begin{pmatrix} u_1^I \\ u_2^I \\ \cdots \\ u_n^I \end{pmatrix} \otimes_i (t_1 \ t_2 \ \ldots \ t_n)) \otimes_r \mathbf{A_1^I} \\
&\subseteq (\mathbf{O}^{'} \ominus_r \begin{pmatrix} u_1^I \\ u_2^I \\ \cdots \\ u_n^I \end{pmatrix} \otimes_r (t_1 \ t_2 \ \ldots \ t_n)) \otimes_r \mathbf{A_1^I} \\
&\subseteq \mathbf{O}^{'} \otimes_r \mathbf{A_1^I} \ominus_r (\begin{pmatrix} u_1^I \\ u_2^I \\ \cdots \\ u_n^I \end{pmatrix} \otimes_r (t_1 \ t_2 \ \ldots \ t_n)) \otimes_r \mathbf{A_1^I} \\
&= \mathbf{O}^{'} \otimes_r \mathbf{A_1^I} \ominus_r \begin{pmatrix} u_1^I \\ u_2^I \\ \cdots \\ u_n^I \end{pmatrix} \otimes_r ((t_1 \ t_2 \ \ldots \ t_n) \otimes_r \mathbf{A_1^I}) \\
&\subseteq \mathbf{O}^{'} \otimes \mathbf{A_1^I} \ominus \begin{pmatrix} u_1^I \\ u_2^I \\ \cdots \\ u_n^I \end{pmatrix} \otimes ((t_1 \ t_2 \ \ldots \ t_n) \otimes \mathbf{A_1^I})
\end{aligned}$$

Similarly, we can show

$$\mathbf{O_{new}} \otimes_r \mathbf{A^I_2} \quad \subseteq \quad \mathbf{O}' \otimes \mathbf{A^I_2} \ominus \begin{pmatrix} u_1^I \\ u_2^I \\ \dots \\ u_n^I \end{pmatrix} \otimes ((t_1 \; t_2 \; \dots \; t_n) \otimes \mathbf{A^I_2})$$

$\blacksquare$

Thus, $\mathbf{P}'$ satisfies the criterion. The definition of $\mathbf{O_{new}}$ also explains why we need to widen the components of the $(u_1, \dots, u_n)^T$ vector: this is to facilitate the computation using inward-rounding so that there will be less chance of "rounding inwardly" into empty intervals. Experiments show that each element in the resultant matrix at the right-hand-side of (5.17) contains several floating-point numbers so that the matrix $\mathbf{O_{new}}$ can be easily found. In the case where some elements in the resultant matrix are empty intervals, we can further widen the vector $(u_1, u_2, \dots, u_n)^T$.

Therefore, the preconditioned system is $\mathbf{K^I} \otimes \vec{X^I} = \mathbf{P}' \otimes \vec{b^I}$. The following lemma and theorem show the correctness result of our incremental preconditioning procedure.

Lemma 5.2.10: Given two systems $\mathbf{A^I} \otimes \vec{X^I} = \vec{b^I}$ and $\mathbf{K^I} \otimes \vec{X^I} = \mathbf{P} \otimes \vec{b^I}$. If $\mathbf{P} \otimes_r \mathbf{A^I} \subseteq \mathbf{K^I}$, then the solutions of $\mathbf{A^I} \otimes \vec{X^I} = \vec{b^I}$ are contained in the solutions of $\mathbf{K^I} \otimes \vec{X^I} = \mathbf{P} \otimes \vec{b^I}$.

Proof : Preconditioning guarantees that the solutions of $\mathbf{A^I} \otimes \vec{X^I} = \vec{b^I}$ are contained in the solutions of $\mathbf{P} \otimes_r \mathbf{A^I} \otimes \vec{X^I} = \mathbf{P} \otimes_r \vec{b^I}$. Since $\mathbf{P} \otimes_r \mathbf{A^I} \subseteq \mathbf{K^I}$ and $\mathbf{P} \otimes_r \vec{b^I} \subseteq \mathbf{P} \otimes \vec{b^I}$, from the inclusion monotonicity of interval arithmetic, we know that the solutions of $\mathbf{P} \otimes_r \mathbf{A^I} \otimes \vec{X^I} = \mathbf{P} \otimes_r \vec{b^I}$ are contained in the solutions of $\mathbf{K^I} \otimes \vec{X^I} = \mathbf{P} \otimes \vec{b^I}$. It follows that the solutions of $\mathbf{A^I} \otimes \vec{X^I} = \vec{b^I}$ are contained in the solutions of $\mathbf{K^I} \otimes \vec{X^I} = \mathbf{P} \otimes \vec{b^I}$. $\blacksquare$

Theorem 5.2.11: Assume that we have a linear system with n equalities of n variables. The incremental preconditioned interval Gauss-Seidel method has the worst case complexity $O(n^3)$. The incremental method preserves all solutions of an interval linear system.

Proof : The preconditioner update algorithm is a variant of incremental Gaussian elimination, which has the worst case complexity $O(n^3)$.

The preconditioned system is updated incrementally in $O(n^2)$ time whenever a new constraint is added. In the worst case, only one new equality is collected in each derivation step, the incremental preconditioned interval Gauss-Seidel method has complexity $O(n^3)$.

From lemmas 5.2.9 and 5.2.10, the incremental method preserves all solutions of an interval linear system. $\blacksquare$

Trailing

There is always a tradeoff between storage and speed in implementing trailing. The state before the last choice point can be restored in higher speed if more data are trailed, but more storage is consumed. This phenomenon becomes more visible in CIAL since interval arithmetic is more demanding than floating-point arithmetic in terms of both storage and speed.

We have several kinds of data in the preconditioned interval Gauss-Seidel solver: the original system of constraints, the associated interval bounds for interval variables, the preconditioner, and the preconditioned system. The original constraints will not be changed so they should never be involved in trailing. To maintain certain level of incremental execution in a constraint logic programming language, we trail the interval bounds and the preconditioner. Since the floating-point preconditioner is updated incrementally using a variant of the incremental Gaussian elimination procedure, trailing the preconditioner should not consume much more memory than trailing constraints in CLP($\mathcal{R}$).

Trailing the preconditioned system suffers from a large space utilization. Consider a system with n equalities of n variables. It takes at least 16 bytes[10] to store an interval and thus an equality occupies $16n$ bytes. In the worst case, i.e. when only *one* new equality is collected in each derivation step and the entire preconditioner is modified, we need to trail $n(n+1)/2$ constraints in total. It requires $(n^2(n+1)/128)$KB.

A problem of size $n = 100$ takes about 7MB storage space. Doubling the problem size rapidly increases the space utilization to about 61MB, which is demanding.

In our design of the preconditioned interval Gauss-Seidel linear solver, we do not trail any entry of the preconditioned system. Upon backtracking, the preconditioned constraints will be re-computed if necessary. This design is based on the assumption that *there are usually some new constraints added to the system after backtracking*. It is inevitable to re-apply preconditioning on some constraints even if they are trailed. Thus, the overhead are *reduced*.

5.2.4 Solver Interaction

The following steps replace the steps 2 and 5 in in algorithm 4.2 to yield a complete constraint solving step in CIAL. Preconditioning in the step 2 serves as constraint transformation. The associated intervals of interval variables are narrowed in the step 5.

2. If any rows of the preconditioner are restored from trail in backtracking, we re-apply preconditioning on the corresponding constraints. Otherwise, the preconditioner and the preconditioned system are updated incrementally to include the new linear constraint. All modified preconditioned constraints are appended to the active list LA `(B)`.

5. Assume that we have collected r linear equality constraints of c variables. Remove a preconditioned linear constraint from the active list LA, say the k-th one of the linear system, the value of the k-th variable is updated using equation (5.6); while the $(r+1)$-st to the c-th variables are narrowed by interval narrowing. If any of the variables is changed, the constraints in both solvers that share that variable will be appended to LA and NA accordingly `(D)`.

Theorem 5.2.12: The constraint solving step in algorithm 4.2 with the incremental preconditioned interval Gauss-Seidel solver always terminates. The system is either inconsistent or stable.

Proof : Since the input constraints are finite, the preconditioner and the preconditioned system update procedures will not be invoked infinitely. Interval Gauss-Seidel method can be considered as "partial" interval narrowing (lemma 5.2.5). Since interval narrowing always terminates with an inconsistent or a stable system [36], it follows that the constraint solving step always terminates. The system is also either inconsistent or stable. ∎

5.3 Comparisons

We have presented how the two proposed linear solvers can be adapted for incremental execution. The soundness of the solvers has also been proved. We show their efficiency and accuracy by some expermental results in the next chapter. Before moving on to the benchmarking results, we give some theoretical comparisons between the generalized interval Gaussian elimination solver and the incremental preconditioned interval Gauss-Seidel linear solver.

[10] We assume that the bounds of an interval are stored as double floating-point numbers.

5.3.1 Time Complexity

The incremental preconditioned interval Gauss-Seidel method is of complexity $O(n^3)$. However, when it is incorporated into a linear solver without any preconditioned systems being trailed, the constraint solving procedure of the linear solver has a different complexity. Consider a general system of n equalities with n unknown variables. In the best case, no constraints are retracted in backtracking, the constraint solving procedure shares the same complexity as the incremental preconditioned interval Gauss-Seidel method, which is of complexity $O(n^3)$. In the worst case, some constraints are retracted and all elements in the preconditioner are modified in backtracking. Since no preconditioned systems are trailed, the previous preconditioned system cannot be restored. Thus the preconditioned system cannot be updated incrementally. We need to apply preconditioning from scratch whenever a new constraint is added. The constraint solving procedure is of complexity $O(n^4)$.

Generalized interval Gaussian elimination is based on incremental Gaussian elimination and generalized interval arithmetic. The former technique has the same complexity as Gaussian elimination, which is of $O(n^3)$. Recall that a generalized interval keeps all intervals that it depends on during calculation. The complexity of a generalized interval operation is in $O(n)$ of its counterpart in ordinary interval arithmetic. The complexity of generalized interval Gaussian elimination is $O(n^4)$.

5.3.2 Storage Complexity

Assume that an interval occupies p storage space. We need np storage to store an interval linear equality of n variables. In the case where no trailing is involved, a system of n equalities takes n^2p storage. On the contrary, if we need to trail all the collected constraints whenever a new constraint is added, there are totally $n(n-1)/2$ constraints to store and they occupy $n^2(n-1)p/2$ storage.

In generalized interval Gaussian elimination, a generalized interval coefficient is a list of ordinary intervals. Thus, the storage complexity of the generalized interval Gaussian elimination is of $O(n^3)$ in the best case and of $O(n^4)$ in the worst case.

In the incremental preconditioned interval Gauss-Seidel method, elements of the preconditioner are floating-point numbers. Storing a floating-point preconditioner requires only half of the storage for storing a system of interval constraints. The storage complexity of the incremental preconditioned interval Gauss-Seidel method is of $O(n^2)$ in the best case and of $O(n^3)$ in the worst case. If the preconditioned system is also trailed, five times of the storage is required. The order of complexity remains unchanged.

5.3.3 Others

Besides the time and storage complexities, there exists some minor differences between the two constraint solving methods.

The preconditioned interval Gauss-Seidel method and preconditioned interval Gaussian elimination give similar sharp results [48]. Since preconditioned interval Gaussian elimination always works better than generalized interval Gaussian elimination, we can expect that the incremental preconditioned interval Gauss-Seidel method also works better than generalized interval Gaussian elimination. Both of them are, however, incomplete in detecting inconsistency due to overestimations introduced by outward-rounding. *The solutions given by these two methods should be interpreted as conditional answers.*

For a system where there are as many independent equalities as variables, generalized interval Gaussian elimination can give solutions without requiring any initial bounds on the variables. Interval Gauss-Seidel method is an iterative method. Initial bounds for some variables are inevitable in general[11].

In short, the incremental preconditioned interval Gauss-Seidel method is compared favourably against generalized interval Gaussian elimination.

[11] Initial bounds on variables can still be omitted in some special cases.

	Generalized Interval Gaussian Elimination Solver	Incremental Preconditioned Interval Gauss-Seidel Solver
Soundness	Yes	Yes
Completeness	No	No
Time (Best)	$O(n^4)$	$O(n^3)$
Time (Worst)	$O(n^4)$	$O(n^4)$
Storage (Best)	$O(n^3)$	$O(n^2)$
Storage (Worst)	$O(n^4)$	$O(n^3)$
Accuracy	Low	High
Initial Bounds	Not necessary	Necessary

Table 5.1: A Summary of Comparisons between Two Linear Solvers

Chapter 6

Benchmarkings

In order to demonstrate the feasibility of our proposal, we have constructed several CIAL prototypes using the C programming language. Since CIAL has much in common with CLP($\mathcal{R}$), we decided to use CLP($\mathcal{R}$) as the backbone of our implementation and tried to adopt as much original CLP($\mathcal{R}$) code as possible.

We use ICLP($\mathcal{R}$) as our first prototype implementation. The bounds of the interval variables are expressed as inequality constraints in CLP($\mathcal{R}$). Although CLP($\mathcal{R}$) can express interval constraints well, it fails to narrow the intervals. We embed interval narrowing for solving interval constraints. The resultant prototype is, however, only four to five times faster than the ICLP($\mathcal{R}$) meta-interpreter [38]. To achieve the goal of efficient constraint solving, we decide to re-use only the Prolog engine part of CLP($\mathcal{R}$) in our subsequent CIAL prototype implementations.

We have completed three different CIAL prototypes, all of which are based on CLP($\mathcal{R}$) Version 1.2 [28]. The solver interface and the two solvers (the linear solver and the non-linear solver) are implemented from scratch. We also modify the unification algorithm of the Prolog engine to cope with unification between interval variables and other terms.

CIAL (Alpha) [15], which consists of about 2800 extra lines of C code, employs generalized interval Gaussian elimination in its linear solver. The proposed preconditioned interval Gauss-Seidel method has been incorporated into CIAL 1.x (Beta) [14]. *CIAL 1.0 (Beta)*, which is implemented in about 3000 extra lines of code, is being freely distributed to the public for experimental use, but the solver lacks incremental execution. Although the preconditioner is constructed incrementally, the preconditioned system (multiplying the preconditioner and the interval coefficient matrix) are re-computed at every derivation step. We further embed the incremental preconditioning algorithm in *CIAL 1.1 (Beta)* to improve its efficiency.

The three prototypes use interval narrowing with splitting in solving inequality and non-linear constraints.

In this chapter, we compare our three CIAL prototypes with BNR Prolog v3.1.0 [7, 52], CLP(BNR) (or BNR Prolog v4.2.3) [50, 9], Echidna Version 0.947 beta [55, 54], ICL [37], and CLP($\mathcal{R}$) Version 1.2 [28, 32] over seven numerical examples of various types: the well-known mortgage program [28] which comes with the CLP($\mathcal{R}$) distribution, a simple system of simultaneous equations in two variables, analysis of a simple DC circuit, inconsistent simultaneous equations, the ball collision problem [30], the famous Wilkinson polynomial [30], and two large systems of linear equations. The examples range from purely linear constraints, to a mixture of linear and non-linear constraints, and to purely non-linear constraints. BNR Prolog runs on an Apple Mac II ($\sim$ 2 VAX MIPS with 5MB Ram) and the other systems run on a SUN SPARCstation 10 model 30 ($\sim$ 49 VAX MIPS with 32MB Ram).

We conclude this chapter by giving a comparison of the performance between CIAL 1.0 (Beta) and CIAL 1.1 (Beta). It shows how the efficiency of linear constraint solving is improved by an incremental preconditioning algorithm.

6.1 Mortgage

The mortgage program [28] is a standard example from CLP($\mathcal{R}$) for relating the principal, number of months, interest rate, outstanding balance and monthly payment in a mortgage. We rewrite it in CIAL syntax.

```
mg(P,T,I,B,MP) :-
      T =:= 1,
      B =:= P + (I*P - MP).
mg(P,T,I,B,MP) :-
      T >= 2,
      TA =:= (1 + I)*P - MP,
      TB =:= T - 1,
      mg(TA,TB,I,B,MP).
```

Given the following query,

```
?- mg(99999,10,0.01,B,5000).
```

CLP($\mathcal{R}$) responds as follows:

```
B = 58150.
```

The above answer is unsound due to round-off errors. Echidna suffers from the same problem by giving answer `58150.03`. Upon the same query, all three CIAL prototypes and ICL give a fairly sharp inclusion of the answer,

$$\mathtt{B} \in (\mathtt{58150.0452133925}, \mathtt{58150.0452133929}).$$

CLP(BNR) agrees with our answer with a slightly lower precision. BNR Prolog returns an interval with wide width.

There is no need for constraint solving, linear or non-linear, with the particular query instantiation. The computation involved is straight value propagation. This example illustrates the problem of roundoff error.

6.2 Simple Linear Simultaneous Equations

This example is to exhibit the inadequacy of interval narrowing for handling linear systems. Let us consider the following simple system in two unknowns.

$$\begin{cases} \mathtt{X} + \mathtt{Y} = \mathtt{5} \\ \mathtt{X} - \mathtt{Y} = \mathtt{6} \\ \mathtt{X}, \mathtt{Y} \in (-\infty, +\infty) \end{cases}$$

All of BNR Prolog, CLP(BNR), Echidna, and ICL cannot narrow any of the variables when no initial bound is given. When we give them an initial guess of, say, $\mathtt{X}, \mathtt{Y} \in [-\mathtt{1000}, \mathtt{1000}]$, they return

$$\mathtt{X} \in [-\mathtt{989}, \mathtt{1000}], \mathtt{Y} \in [-\mathtt{995}, \mathtt{994}].$$

All CIAL prototypes[1] return

$$\mathtt{X} \in [5.5, 5.5], \mathtt{Y} \in [-0.5, -0.5].$$

which is the same exact solution returned by CLP($\mathcal{R}$).

It is interesting to note that the first four systems are able to return slightly less accurate solutions when interval splitting is employed.

6.3 Analysis of DC Circuit

Electrical engineering is an important application area for constraint logic programming [27]. Consider the simple DC circuit in figure 6.1. We are interested in the currents passing through the resistors.

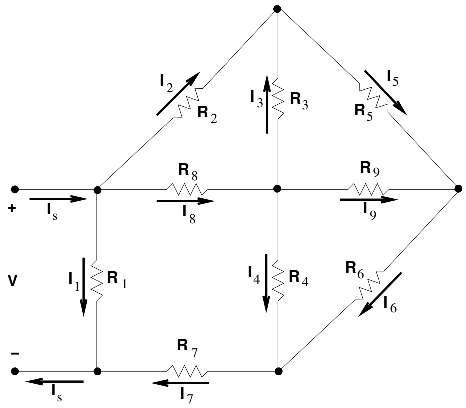

Figure 6.1: A Simple DC Circuit

Assume that $V = 10$ volts and $R_i = i\ \Omega$ for $i = 1, 2, \cdots, 9$. The following system of linear equations are obtained from nodal and mesh analysis.

$$\left\{\begin{array}{rcll} \mathtt{I_s} - \mathtt{I_1} - \mathtt{I_2} - \mathtt{I_8} & = & 0 & , \qquad \mathtt{I_1} = 10 \\ -\mathtt{I_s} + \mathtt{I_1} + \mathtt{I_7} & = & 0 & , \qquad 2\mathtt{I_2} - 3\mathtt{I_3} - 8\mathtt{I_8} = 0 \\ \mathtt{I_2} + \mathtt{I_3} - \mathtt{I_5} & = & 0 & , \qquad 3\mathtt{I_3} + 5\mathtt{I_5} - 9\mathtt{I_9} = 0 \\ -\mathtt{I_3} - \mathtt{I_4} + \mathtt{I_8} - \mathtt{I_9} & = & 0 & , \qquad -4\mathtt{I_4} + 6\mathtt{I_6} + 9\mathtt{I_9} = 0 \\ \mathtt{I_4} + \mathtt{I_6} - \mathtt{I_7} & = & 0 & , \qquad -\mathtt{I_1} + 4\mathtt{I_4} + 7\mathtt{I_7} + 8\mathtt{I_8} = 0 \\ \mathtt{I_5} - \mathtt{I_6} + \mathtt{I_9} & = & 0 & \end{array}\right.$$

There are 11 linear equations but only 10 unknown variables. The redundant equation cannot be located in advance, however. CIAL (Alpha) gives the following results in 1.11s.

$$\begin{array}{rcl} \mathtt{I_s} & \in & (10.8282985772, 10.8282985773) \\ \mathtt{I_1} & \in & [10.0000000000, 10.0000000000] \\ \mathtt{I_2} & \in & (0.5690898460, 0.5690898461) \\ \mathtt{I_3} & \in & (-0.3118300527, -0.3118300526) \end{array}$$

[1] CIAL 1.0 (Beta) and CIAL 1.1 (Beta) use iterative method (Gauss-Seidel method) to solve constraints after symbolic pre-processing (preconditioning). Initial bounds for some variables are usually necessary, although they can be omitted in this example.

$$\begin{array}{lcl} \mathtt{I}_4 & \in & (0.5320600272, 0.5320600273) \\ \mathtt{I}_5 & \in & (0.2572597934, 0.2572597935) \\ \mathtt{I}_6 & \in & (0.2962385499, 0.2962385500) \\ \mathtt{I}_7 & \in & (0.8282985772, 0.8282985773) \\ \mathtt{I}_8 & \in & (0.2592087312, 0.2592087313) \\ \mathtt{I}_9 & \in & (0.0389787565, 0.0389787566) \end{array}$$

CLP($\mathcal{R}$) responds in less than 1/60 second. CLP($\mathcal{R}$)'s efficiency over CIAL (Alpha) is due to the fact that interval symbolic operations, i.e. interval forward and backward substitutions, are time-consuming. The solutions given by CLP($\mathcal{R}$) are, however, unsound.

With initial value `[-100,100]` for all variables, both CIAL 1.0 (Beta) and CIAL 1.1 (Beta) give the same results as in CIAL (Alpha) and in less than 1/60 second.

By splitting 4 variables ($\mathtt{I}_s, \mathtt{I}_1, \mathtt{I}_2, \mathtt{I}_3$), ICL exits abruptly after 2 minutes of execution; Echidna (in high precision) and BNR Prolog cannot terminate in 2 and 24 hours respectively. CLP(BNR) cannot give any solution (except $\mathtt{I}_1$) with width less than 100, although all variables are specified to split.

This example demonstrates the inability of interval narrowing with splitting to solve even small systems of linear constraints.

6.4 Inconsistent Simultaneous Equations

The following ad hoc constraint system is obviously inconsistent, since `B` and `C` should be equal with value either 1 or -2 according to the first three constraints. Neither value is, however, consistent with the initial bound of `B`.

$$\begin{cases} \mathtt{A}+\mathtt{C} & = & \mathtt{D} \\ \mathtt{A}+\mathtt{B} & = & \mathtt{D} \\ \mathtt{C}(\mathtt{C}+1) & = & 2 \\ \mathtt{A} \in (0,\infty), \mathtt{B} \in (-\infty,-5) \end{cases}$$

With interval splitting on all variables, CLP(BNR) returns "yes;" ICL and BNR Prolog do not terminate in 1 hour; Echidna returns "yes" with default precision and exits abruptly with high precision. CLP($\mathcal{R}$) gives "maybe" with answer constraints.

All three CIAL prototypes can detect the inconsistency without using splitting but with the cooperation of two solvers.

It is interesting to find that given the constraint `C (C + 1) = 2`, none of CLP(BNR), BNR Prolog, and Echidna can calculate the value of `C` without using splitting, even initial guess `[-100,100]` is given.

6.5 Collision Problem

We demonstrate the non-linear constraint solving ability of CIAL in the two subsequent examples.

This collision problem and the following program are adopted from [30]. The program describes two objects, one stationary cubic wall and a ball moving along a quadratic space curve. It tries to find the time that the ball hits the wall.

```
% object_A/3 describes the shape of the wall
object_A(X,Y,Z) :-
      X <= 0,
      Y <= 0,
      Z <= 0.
% object_B/3 describes the shape of the ball
object_B(X,Y,Z) :-
      X^2 + Y^2 + Z^2 =:= 1.
% center_B/4 gives the position of the center of the ball
% at time T
center_B(T,Cx,Cy,Cz) :-
      Cx =:= T^2 - 10,
      Cy =:= 2*T - 10,
      Cz =:= T^2 - 7*T + 10.
% object_B_moving/4 gives the point (X,Y,Z) that is in the
% ball at time T
object_B_moving(T,X,Y,Z) :-
      center_B(T,Cx,Cy,Cz),
      object_B(X-Cx,Y-Cy,Z-Cz).
```

Given the following query,

```
?- T >= 0, object_A(X,Y,Z), object_B_moving(T,X,Y,Z).,
```

all CIAL prototypes give the result,

$$\mathtt{T} \in (1.6972243622, 3.3166247904).$$

It is the same as the results obtained from RISC-CLP(Real) [30], which employs symbolic algebraic method for constraint solving.

$$\begin{cases} \mathtt{T} <= 3, \\ \mathtt{T}^2 - 7\mathtt{T} + 9 <= 0. \end{cases} \quad \text{or} \quad \begin{cases} \mathtt{T} >= 3, \\ \mathtt{T}^2 - 11 <= 0. \end{cases}$$

RISC-CLP(Real) cannot solve the above quadratic equations. We use some algebra packages to solve the two systems. The union of the answers are

$$\mathtt{T} \in (1.6972243622, 3.3166247904).$$

CIAL cannot give this sharp result if we do not use the `square` primitive constraint.

Both BNR Prolog, CLP(BNR), and ICL return similar results as CIAL. This is predictable since their solvers are also based on interval narrowing. Echidna returns a wide answer at low precision and exits abruptly at higher precision.

For efficiency reason, CLP($\mathcal{R}$) does not provide non-linear constraint solving. All non-linear arithmetic constraints are classified as hard constraints, which will be considered only when they become linear [44]. In this example, since no non-linear constraints can become linear, they are delayed indefinitely. The output of CLP($\mathcal{R}$) is

```
0 <= T,
X <= 0,
Y <= 0,
Z <= 0,
-_t12 * _t12 - (Y - 2*T + 10) * (Y - 2*T + 10) + 1 = _t10 * _t10,
-_t12 + Z + 7*T - 10 = T * T,
-_t10 + X + 10 = T * T.
```

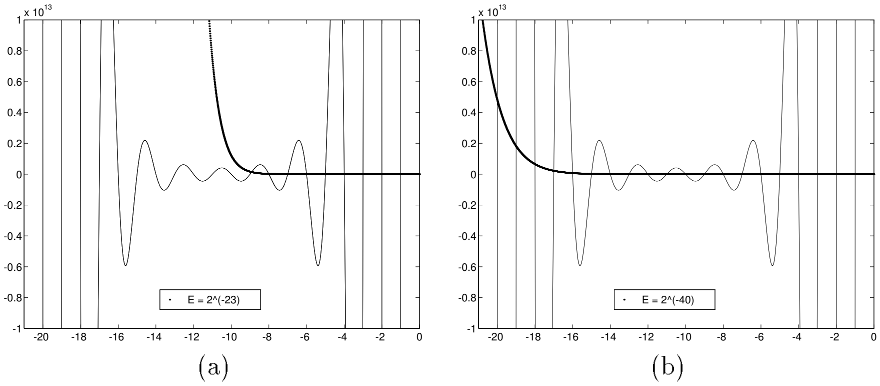

Figure 6.2: The curves $y = \prod_{i=1}^{20}(X+i)$ and $y = -EX^{19}$

6.6 Wilkinson Polynomial

This example describes the famous Wilkinson polynomial equation. The problem is to find the real roots of the following equation.

$$\prod_{i=1}^{20}(X+i) + EX^{19} = 0$$

Let $E = 0$. The real roots of this unperturbed polynomial in the closed interval $[-20, -10]$ are -20,-19,-18,-17,-16,-15,14,-13,-12,-11,-10, respectively.

A slight perturbation of the polynomial by $E = 2^{-23}$ removes all roots in $[-20, -10]$, as shown in figure 6.2(a) which consists of the curves $Y = \prod_{i=1}^{20}(X+i)$ and $Y = -EX^{19}$. CIAL returns no answers as expected.

When $E = 2^{-40}$ (Figure 6.2(b)), the three CIAL prototypes find all solutions with 10 decimal place accuracy by using interval splitting.

$$
\begin{array}{lcl}
\mathtt{X} & \in & (-10.0000069069\ldots, -10.0000069068\ldots) \\
\mathtt{X} & \in & (-10.9999577619\ldots, -10.9999577618\ldots) \\
\mathtt{X} & \in & (-12.0001805794\ldots, -12.0001805793\ldots) \\
\mathtt{X} & \in & (-12.9994495357\ldots, -12.9994495356\ldots) \\
\mathtt{X} & \in & (-14.0012132530\ldots, -14.0012132529\ldots) \\
\mathtt{X} & \in & (-14.9980738185\ldots, -14.9980738184\ldots) \\
\mathtt{X} & \in & (-16.0021894094\ldots, -16.0021894093\ldots) \\
\mathtt{X} & \in & (-16.9982666582\ldots, -16.9982666581\ldots) \\
\mathtt{X} & \in & (-18.0009047580\ldots, -18.0009047579\ldots) \\
\mathtt{X} & \in & (-18.9997188357\ldots, -18.9997188356\ldots)
\end{array}
$$

Echidna cannot terminate in 15 minutes for $E = 2^{-23}$. For $E = 2^{-40}$, it aborts abnormally for default precision and does not give any solution in 15 minutes for higher precision. ICL, BNR Prolog, and CLP(BNR) return similar results as those of CIAL in both cases. CLP($\mathcal{R}$) delays all the constraints and gives

```
X <= -10,
-20 <= X,
-E * X * X * X * X * X * X * X * X * X * X * X * X * X *
   X * X * X * X * X * X = (X + 1) * (X + 2) * (X + 3) *
   (X + 4) * (X + 5) * (X + 6) * (X + 7) * (X + 8) * (X + 9) *
   (X + 10) * (X + 11) * (X + 12) * (X + 13) * (X + 14) *
   (X + 15) * (X + 16) * (X + 17) * (X + 18) * (X + 19) *
   (X + 20).
```

6.7 Summary and Discussion

The last example on large-scale linear systems is beyond the capability of the interval narrowing based systems, i.e. ICL, BNR Prolog, CLP(BNR), and Echidna. We compare only the three CIAL prototypes and CLP($\mathcal{R}$). Before going into the last example, we summarize the previous benchmarking results in table 6.1 and table 6.2, and give a brief discussion.

The CIAL solvers subsume the symbolic constraint solving method (Gaussian elimination or preconditioning) and the interval narrowing technology. CLP($\mathcal{R}$) delays non-linear constraints from consideration and interval narrowing fails to handle even small system of linear constraints, as shown in section 6.3. Thus, CLP($\mathcal{R}$), BNR Prolog, CLP(BNR), Echidna, and ICL are deficient in solving mixtures of linear and non-linear constraints. A simple example can be obtained by adding the constraint $\mathtt{I_X}(\mathtt{I_X} - \mathtt{1}) = \mathtt{I_S}$ to the system in section 6.3.

6.8 Large System of Simultaneous Equations

The following program describes a randomly generated system of linear constraints with rank=50.

```
test(X0,X1,X2,...,X49) :-
      a random generated constraint with 50 variables,
      fail.
t :-
      X0>= -10000,X0<=10000,
      X1>= -10000,X1<=10000,
         ⋮       ⋮
      X49>= -10000,X49<=10000,
      5 random generated constraints with 50 variables,
      p1(X0,X1,X2,...,X49).
p1(X0,X1,X2,...,X49) :-
      test(X0,X1,X2,...,X49).
p1(X0,X1,X2,...,X49) :-
      5 random generated constraints with 50 variables,
      p2(X0,X1,X2,...,X49).
⋮        ⋮        ⋮
p9(X0,X1,X2,...,X49) :-
      test(X0,X1,X2,...,X49).
p9(X0,X1,X2,...,X49) :-
      5 random generated constraints with 50 variables.
```

The program contains predicates `test/50`, `t/0`, `p1/50`, ..., `p9/50`. The top level predicate of the program is `t/0`, which sets the initial bounds of all variables and calls `p1/50`. Each subsequent `pi/50` predicate

	CLP(BNR)	BNR Prolog v3.1.0	ICL	Echidna
Mortgage	sound result	sound wide result [58149,58151]	sound result	unsound result 58150.03
Simple System	almost no narrowing	almost no narrowing	almost no narrowing	almost no narrowing
Simple System with Splitting	sound result	sound result	sound result	sound result
DC Circuit	almost no narrowing	almost no narrowing	almost no narrowing	almost no narrowing
DC Circuit with splitting	sound but very wide result	not terminate in 24 hours	exit abruptly	not terminate in 2 hours
Inconsistent System	cannot detect	not terminate in 1 hour	not terminate in 1 hour	cannot detect
Collision	sound result	sound result	sound result	exit abruptly at high precision & give wide result at low precision
Wilkinson (2^{-23})	no solution	no solution	no solution	not terminate in 15 minutes
Wilkinson (2^{-40})	sound result	sound result	sound result	exit abruptly at default precision & not terminate in 15 minutes at high precision

Table 6.1: A Summary of Comparisons

definition consists of two clauses. The first clause calls the `test/50` predicate, which submits a new constraint to the constraint solver and always fails. This is to exercise the trailing and backtracking capability of CIAL. The second clause submits five extra constraints to the solver and calls predicate `p(i+1)/50`. That means that the call patterns of the `pi/50` predicates form a "chain" and five constraints are added to the constraint solver in each derivation step.

CIAL 1.0 (Beta) gives the solutions with 10 decimal place accuracy in 34.3s, while CIAL 1.1 (Beta) solves the system with the same precision in 36.2s. CLP($\mathcal{R}$) responds in 3.7s. It is about ten times faster than CIAL since interval computation is time-consuming, especially for setting IEEE rounding directions and detecting exceptions. The solutions given by CLP($\mathcal{R}$) are, however, unsound. Many of them are different from the real solutions at the fourth or fifth decimal place. It is also interesting to find that CIAL (Alpha) cannot narrow any of the variables due to the growth of width of the coefficients.

We further try to solve a larger system. It is similar to the previous one but there are 100 linear constraints in total. Constraints are added to the constraint solver two at a time in each derivation step.

With 10 decimal place accuracy, CIAL 1.0 (Beta) and CIAL 1.1 (Beta) respond in 1312.6s and 1091.2s respectively. CLP($\mathcal{R}$) solves the system in 104.2s. Again, CIAL (Alpha) cannot narrow any of

	CIAL (Alpha)	CIAL 1.0 (Beta)	CIAL 1.1 (Beta)	CLP($\mathcal{R}$)
Mortgage	sound result	sound result	sound result	unsound result 58150
Simple System	sound result	sound result	sound result	sound result
Simple System with Splitting	N/A	N/A	N/A	N/A
DC Circuit	give sound result in 1.11s	give sound result in ~1/60s	give sound result in ~1/60s	give unsound result in ~1/60s
DC Circuit with splitting	N/A	N/A	N/A	N/A
Inconsistent System	can detect	can detect	can detect	cannot detect
Collision	sound result	sound result	sound result	floundering
Wilkinson (2^{-23})	no solution	no solution	no solution	floundering
Wilkinson (2^{-40})	sound result	sound result	sound result	floundering

Table 6.2: A Summary of Comparisons (cont.)

the variables.

This example demonstrates the efficiency and practicality of the incremental preconditioned interval Gauss-Seidel method in the CIAL linear solver. Also, it shows that generalized interval Gaussian elimination fails to tackle large systems of linear equations effectively.

6.9 Comparisons Between the Incremental and the Non-Incremental Preconditioning

We end this chapter by comparing the performance of CIAL 1.0 (Beta) and CIAL 1.1 (Beta). The only difference between these two systems is that the latter employs an incremental preconditioning technique in its linear solver, while the former does not.

We modify the program in section 6.8. The first clause in all the `pi/n` predicate definitions are removed. It implies that no trailing or backtracking will be involved in the execution of the program. Also, the program is changed to only one constraint can be collected in a derivative step.

Table 6.3 shows the speedup in interval linear constraint solving with incremental preconditioning for problem size ranging from 10 to 100. As expected, CIAL 1.1 exhibits a near linear speedup over CIAL 1.0 as the problem size grows.

Solutions given by CIAL 1.1 are slightly wider than those obtained from CIAL 1.0. We find that the solutions given by CIAL 1.1, however, still reach 8 decimal places of accuracy in general.

Rank	CIAL 1.0 (Beta)	CIAL 1.1 (Beta)	Speedup	CLP($\mathcal{R}$)
10	0.35s	0.35s	1.00	0.01s
20	2.87s	2.20s	1.30	0.08s
30	12.99s	8.28s	1.57	0.35s
40	38.50s	24.33s	1.58	0.93s
50	85.06s	48.94s	1.74	2.51s
60	171.73s	92.74s	1.85	6.19s
70	371.58s	198.71s	1.87	8.93s
80	544.25s	283.93s	1.92	17.52s
90	838.93s	417.11s	2.01	34.56s
100	1259.81s	607.91s	2.07	39.07s

Table 6.3: Speedup by Using the Incremental Preconditioning Algorithm

Chapter 7

Concluding Remarks

7.1 Summary and Contributions

In this thesis, we have discussed the deficiencies of interval narrowing with splitting. Our experiments show that interval narrowing based systems fail to solve even small problems efficiently and effectively. Thus interval narrowing with splitting is impractical in solving general interval constraints over real domain (Chapter 3). We propose to separate linear equality constraint solving from inequality and non-linear constraint solving. This idea is realized in our new interval constraint logic programming system, CIAL (for *Constraint Interval Arithmetic Language*), which shares the same declarative and operational semantics as those of ICLP($\mathcal{R}$) [38] (Chapter 2). We have designed an architecture for CIAL and established the interaction among the modules in the architecture. Unification between interval variables and other terms are handled in an extended unification algorithm. Input arithmetic constraints are decomposed into linear equalities and a set of convex primitive constraints. The former is handled by the linear solver; while we apply interval narrowing on the latter in the non-linear solver (Chapter 4). We have extended and generalized two linear constraint solving techniques in interval computation for interval linear equality constraint solving, resulting in generalized interval Gaussian elimination and the incremental preconditioned interval Gauss-Seidel method. These techniques have been implemented in the CIAL linear constraint solvers (Chapter 5). We have constructed three CIAL prototypes with different linear solvers and compared them with several major interval constraint logic programming languages. The performance of the different prototypes are presented (Chapter 6).

The contribution of our work is three-fold. First, we have derived two practical interval linear equality solvers. The solvers are adapted for incremental execution. Their correctness have also been established. Of the two proposed linear constraint solving methods, the incremental preconditioned interval Gauss-Seidel method is of $O(n^3)$ complexity, which is the same as the ordinary Gaussian elimination in solving system of equations. Solutions given by the incremental preconditioned interval Gauss-Seidel method are, however, slightly wider than those obtained from the non-incremental method. Our large scale experiments show that the solutions given by this incremental method still reach 8 decimal places of accuracy in general.

Second, we have shown how an interval linear solver can be incorporated into a system which has already had a non-linear solver based on interval narrowing. We have derived a constraint decomposition procedure and an interaction scheme for the two solvers. Input constraints are divided into two categories, which will be sent to two solvers accordingly. The two solvers share common variables, interact in a round-robin fashion, and cooperate towards solving a system of numerical constraints. We have shown the termination of the interaction scheme.

Third, we have constructed three prototypes of CIAL and compared them with one another, as well as with several existing interval constraint logic programming languages. Of the three prototypes, CIAL 1.1 (Beta) has been shown to be the most efficient one in solving large scale linear systems. On the comparisons of CIAL and other existing systems, CIAL is all rounded: all prototypes are substantially more efficient and can solve more classes of problems than any other existing systems when used alone.

7.2 Future Work

A number of questions remain to be investigated. First, the linear solver can only handle linear equalities. It would be interesting to investigate how linear inequalities can be accommodated. We believe that our proposed linear equality constraint solving methods, especially the incremental preconditioned interval Gauss-Seidel method, can be generalized to handle inequalities. Second, Benhamou *et al* [8] replace interval narrowing by a Newton reduction operator, which shows an improvement in non-linear constraint solving. However, preconditioning has not been included. It is worthwhile to study if our incremental preconditioning technique can be applied to further improve the Newton reduction operator.

On the theoretical side, it would be interesting to study the level of interval consistency attainable in generalized interval Gaussian elimination and the preconditioned interval Gauss-Seidel method. Both of them should reach a consistency level falling between box consistency and hull consistency [8].

Concerning implementations, our CIAL prototypes have much to be desired. First, the CIAL architecture is rudimentary. Further optimizations, such as the techniques used for CLP($\mathcal{R}$), might be applicable to CIAL. Second, the current prototypes implement constraint solvers as independent modules separating from the Prolog engine. Communications between the solvers and the Prolog engine incur high overhead. Backtracking also becomes a costly operation. We expect that the work of Lee and Lee [37] can be used as basis to integrate the interval constraint solving and the Prolog engine at the Warren Abstract Machine (WAM) level. Third, built-in predicates in CIAL are limited. To apply CIAL on real-life problems (e.g. scheduling), more relations, such as `max/2`, `min/2`, `sin/1`, `asin/1` [50], should be provided.

To establish the practicality of our approach, we need to try CIAL on more real-life applications, e.g. job shop scheduling, process planning, assembly line balancing, temporal and spatial reasoning, multiagent planning, finite element modeling, circuit design, etc.

Bibliography

[1] A. Aggoun and N. Beldiceanu. Overview of the CHIP compiler system. In *Proceedings of the Eighth International Conference on Logic Programming*, Paris, France, 1991.

[2] A. Aiba, K. Sakai, Y. Sato, D.J. Hawley, and R. Hasegawa. Constraint logic programming language CAL. In *Proceedings of the International Conference on Fifth Generation Computer Systems 1988*, pages 263–276, Tokyo, Japan, 1988.

[3] H. Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction.* The MIT Press, 1991.

[4] G. Alefeld and J. Herzberger. *Introduction to Interval Computations.* Academic Press, 1983.

[5] K.E. Atkinson. *An Introduction to Numerical Analysis.* John Wiley & Sons, 1978.

[6] R. Barrett, M. Berry, T. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods.* Society for Industrial & Applied Mathematics, 1993.

[7] Bell-Northern Research Ltd. *BNR Prolog Reference Manual*, 1988.

[8] F. Benhamou, D. McAllester, and P. Van Hentenryck. CLP(Intervals) revisited. In *Logic Programming: Proceedings of the 1994 International Symposium*, Ithaca, USA, 1994.

[9] F. Benhamou and W.J. Older. Applying interval arithmetic to real, integer and boolean constraints. *(Submitted to) Journal of Logic Programming*, 1994.

[10] C. Bessiere. Arc-consistency and arc-consistency again. *AI Journal*, 65(1):179–190, 1994.

[11] B. Buchberger. Gröbner bases: An algorithmic method in polynomial ideal theory. In N.K. Bose, editor, *Recent Trends in Multidimensional Systems Theory*, chapter 6. D. Riedel Publ. Comp., 1983.

[12] B. Buchberger and H. Hong. Speeding-up quantifier elimination by Gröbner bases. Technical Report 91-06.0, Research Institute for Symbolic Computation, Johannes Kepler University, A-4040 Linz, Austria, 1991.

[13] X. Chen and D. Wang. On the optimal properties of the Krawczyk-type interval operator. *International Journal of Computer Mathematics*, 29(2-4):235–245, 1989.

[14] C.K. Chiu and J.H.M. Lee. Interval linear constraint solving using the preconditioned interval Gauss-Seidel method. In *Workshop on Constraint Languages/Systems and their use in Problem Modelling*, Ithaca, USA, 1994.

[15] C.K. Chiu and J.H.M. Lee. Towards practical interval constraint solving in logic programming. In *Logic Programming: Proceedings of the 1994 International Symposium*, Ithaca, USA, 1994.

[16] J.G. Cleary. Logical arithmetic. *Future Computing Systems*, 2(2):125–149, 1987.

[17] A. Colmerauer. An introduction to Prolog III. *Communications of the ACM*, 33(7):69–90, July 1990.

[18] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. The MIT Press, eighth edition, 1992.

[19] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The constraint logic programming language CHIP. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, Tokyo, Japan, 1988.

[20] D.M. Gay. Solving interval linear equations. *SIAM Journal on Numerical Analysis*, 19(4):858–870, 1982.

[21] W.W. Hager. Updating the inverse of a matrix. *SIAM Review*, 31(2):221–239, June 1989.

[22] E. Hansen. *Global Optimization using Interval Analysis*. Marcel Dekker, Inc., 1992.

[23] E.R. Hansen. Interval arithmetic in matrix computations. *SIAM Journal on Numerical Analysis*, 2:308–320, 1965.

[24] E.R. Hansen. A generalized interval arithmetic. In *Interval Mathematics*, pages 7–18, 1975.

[25] E.R. Hansen. Bounding the solution of interval linear equations. *SIAM Journal on Numerical Analysis*, 29(5):1493–1503, October 1992.

[26] W.S. Havens, S. Sidebottom, J. Jones, M. Cuperman, and R. Davison. Echidna constraint reasoning system: Next-generation expert system technology. Technical Report CSS-IS TR 90-09, Centre for Systems Science, Simon Fraser University, Burnaby, B.C., Canada, 1990.

[27] N. Heintze, S. Michaylov, and P. Stuckey. CLP($\mathcal{R}$) and some electrical engineering problems. *Journal of Automated Reasoning*, 9(2):231–260, October 1992.

[28] N.C. Heintze, J. Jaffar, S. Michaylov, P.J. Stuckey, and R.H.C. Yap. *The CLP($\mathcal{R}$) Programmer's Manual Version 1.2*. IBM Thomas J Watson Research Center, 1992.

[29] H. Hong. *Improvements in CAD-Based Quantifier Elimination*. PhD thesis, The Ohio State University, 1990.

[30] H. Hong. Non-linear real constraints in constraint logic programming. In *Proceedings of the Second International Conference on Algebraic and Logic Programming*, 1992.

[31] J. Jaffar and J-L. Lassez. Constraint logic programming. *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages*, 1987.

[32] J. Jaffar, S. Michaylov, P.J. Stuckey, and R.H.C. Yap. The CLP($\mathcal{R}$) language and system. In *ACM Transactions on Programming Languages and Systems*, volume 14, pages 339–395, 1992.

[33] R.B. Kearfott. Preconditioners for the interval Gauss-Seidel method. *SIAM Journal on Numerical Analysis*, 27(3):804–822, June 1990.

[34] R.B. Kearfott, C. Hu, and M. Novoa. A review of preconditioners for the interval Gauss-Seidel method. *Interval Computations*, 1(1):59–85, 1991.

[35] R.B. Kearfott and Xing Z. An interval step control for continuation methods. *SIAM Journal on Numerical Analysis*, 31(3):892–914, June 1994.

[36] J.H.M. Lee. *Numerical Computation As Deduction In Constraint Logic Programming*. PhD thesis, Department of Computer Science, Logic Programming Laboratory, University of Victoria, Victoria, Canada, 1992.

[37] J.H.M. Lee and T.W. Lee. A WAM-based abstract machine for interval constraint logic programming. In *Proceedings of the Sixth IEEE International Conference on Tools with Artificial Intelligence*, New Orleans, USA, 1994.

[38] J.H.M. Lee and M.H. van Emden. Adapting CLP($\mathcal{R}$) to floating-point arithmetic. In *Proceedings of the International Conference on Fifth Generation Computer Systems 1992*, volume 16, pages 996–1003, Tokyo, Japan, 1992.

[39] J.H.M. Lee and M.H. van Emden. Interval computation as deduction in CHIP. *Journal of Logic Programming*, 16:255–276, 1993.

[40] O. Lhomme. Consistency techniques for numeric CSPs. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence*, 1993.

[41] J.W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, second edition, 1987.

[42] A.K. Mackworth. Consistency in networks of relations. *AI Journal*, 8(1):99–118, 1977.

[43] A.K. Mackworth, J.A. Mulder, and W.S. Havens. Hierarchical arc consistency: Exploiting structured domains in constraint satisfaction problems. *Computational Intelligence*, 1:118–126, 1985.

[44] S. Michaylov. *Design and Implementation of Practical Constraint Logic Programming Systems.* PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, U.S.A, August 1992.

[45] R.E. Moore. *Interval Analysis.* Prentice-Hall, 1966.

[46] J.Ll. Morris. *Computational Methods in Elementary Numerical Analysis.* John Wiley & Sons, 1983.

[47] A. Neumaier. Overestimation in linear interval equations. *SIAM Journal on Numerical Analysis*, 24(1):207–214, February 1987.

[48] A. Neumaier. *Interval Methods for Systems of Equations.* Cambridge University Press, 1990.

[49] Members of the Radix-Independent Floating-point Arithmetic Working Group. IEEE standard for radix-independent floating-point arithmetic. Technical Report ANSI/IEEE Std 854-1987, The Institute of Electrical and Electronics Engineers, New York, USA, 1987.

[50] W. Older. Constraints in BNR Prolog. Technical Report Draft 01, Software Engineering Centre, Bell-Northern Research, Ottawa, Canada, 1993.

[51] W. Older and A. Vellino. Extending Prolog with constraint arithmetic on real intervals. In *Proceedings of the Canadian Conference on Computer & Electrical Engineering*, Ottawa, Canada, 1990.

[52] W. Older and A. Vellino. Constraint arithmetic on real intervals. In A. Colmerauer and F. Benhamou, editors, *Constraint Logic Programming: Selected Research.* MIT Press, 1992.

[53] W.J. Older. The application of relational arithmetic to X-ray diffraction crystallography. Technical Report 89001, Software Engineering Centre, Bell-Northern Research, Ottawa, Canada, 1989.

[54] G. Sidebottom and W.S. Havens. Hierarchical arc consistency for disjoint real intervals in constraint logic programming. *Computational Intelligence*, 8(4):601–623, 1992.

[55] S. Sidebottom, W. Havens, and S. Kindersley. *Echidna Constraint Reasoning System (Version 1): Programming Manual.* Expert Systems Laboratory, Simon Fraser University, British Columbia, Canada, 2.0 edition, 1992.

[56] L. Sterling and E. Shapiro. *The Art of Prolog.* The MIT Press, second edition, 1994.

[57] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming*. The MIT Press, London, England, 1989.

[58] J.H. Wilkinson. *The Algebraic Eigenvalue Problem*. Oxford University Press, 1965.